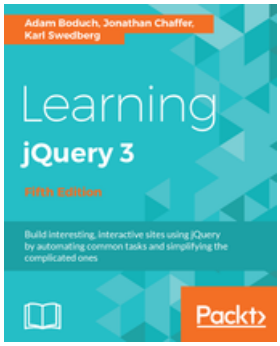

Chapter 3

Handling Events

Timing of code execution

$(() => {}) was jQuery's primary way to perform tasks on page load. It is not, however, the only method at our disposal. The window.onload event fires when a document is completely downloaded to the browser. Each handler registered using $(() => {}) is invoked when the DOM is completely ready for use. Difference is window.onload requires every element on the page to be ready and the jQuery $(() => {}) method only requires the HTML content to be ready.

Handling multiple scripts on one page

Old style of assigning event handlers:

```
function doStuff() {
    // Perform a task...
}

<body onload="doStuff();">   or    window.onload = doStuff;
```

With a single function, this is easy. But with multiple functions, as we saw in 263, it got out of hand. Instead, $(() => {}) mechanism handles this situation gracefully. Each call adds the new function to an internal queue of behaviors.

Passing an argument to the document ready callback

In some cases, it may prove useful to use more than one JavaScript library on the same page and many may use the $. jQuery provides a method called jQuery.noConflict() to return control of the $ identifier back to other libraries.

```
<script src="prototype.js"></script>
  <script src="jquery.js"></script>
  <script>
   jQuery.noConflict();
  </script>
```

```
<script src="myscript.js"></script>
```

First, the prototype.js library is included. Then, jquery.js is included, taking over $ for its own use. Next, a call to .noConflict() frees up $, so that control of it reverts to the first included library (prototype.js) within myscript.js so that it will use both libraries.

A simple style switcher

HTML:

```
<div id="switcher" class="switcher">
    <h3>Style Switcher</h3>
    <button id="switcher-default">
      Default
    </button>
    <button id="switcher-narrow">
      Narrow Column
    </button>
    <button id="switcher-large">
      Large Print
    </button>
</div>
```

CSS:

```
body.large .chapter {
    font-size: 1.5em;
}
```

Grab the rest of the source from the source files for your book.

jQuery:

```
$(() => {
    $('#switcher-large')
     .on('click', () => {
       $('body').addClass('large');
}); });
```

That's all there is to binding a behavior to an event. Multiple calls to .on() coexist nicely

ALL BUTTONS:

CSS:

```
 body.narrow .chapter {
   width: 250px;
```

```
}
```

jQuery

```
$(() => {
  $('#switcher-default')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .removeClass('large');
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow')
        .removeClass('large');
});

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .addClass('large');
}); });
```

And to add functionality to let the user know which button is selected:

CSS:

```
.selected {
    font-weight: bold;
}
```

When any event handler is triggered, the keyword this refers to the DOM element to which the behavior was attached. By writing $(this) within the event handler, we create a jQuery object corresponding to the element, and we can act on it just as if we had located it with a CSS selector.

```
$(this).addClass('selected');
```

Placing this line in each of the three handlers will add the class when a button is clicked, or even more streamlined, see below.

To remove the class from the other buttons, we can take advantage of jQuery's implicit

iteration feature, and write:

```
$('#switcher button').removeClass('selected');
```

Complete, refactored code. Refactoring means to modify existing code to perform the same task in a more efficient or elegant way. Note that the order of operations has changed a bit to accommodate our more general class removal; we need to execute .removeClass() first so that it doesn't undo the call to .addClass()

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      $('body')
        .removeClass();
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow');
});

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .addClass('large');
});
});
```

First, **implicit iteration** is once again useful when we bind the same click handler to each button with a single call to .on(). Second, **behavior queuing** allows us to bind two functions to the same click event without the second overwriting the first.

Finally, we can get rid of the specific handlers entirely by, once again, exploiting **event context**.

```
$(() => {
  $('#switcher-default')
```

```
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      const bodyClass = this.id.split('-')[1];
      $('body')
        .removeClass()
        .addClass(bodyClass);
      $('#switcher button')
        .removeClass('selected');
      $(this)
          .addClass('selected');
  });
});
```

If you only got to one of the levels above for this course and your code worked, I would not count it against you. If you can completely refactor your code as to what is immediately above, congratulations, but sometimes, this takes years to master as you need practice.

Shorthand Events

Binding a handler for an event (such as a simple click event) is such a common task that jQuery provides an even terser way to accomplish it; shorthand event methods work in the same way as their .on() counterparts with fewer keystrokes. As an example, could replace the on event above with a click:

```
… $('#switcher button')
    .click(function() {…
```

Showing and hiding page elements

CSS for hiding buttons:

```
.hidden {
    display: none;
}
```

jQuery showing the toggleClass method()

```
$(() => {
  $('#switcher h3')
    .click(function() {
      $(this)
        .siblings('button')
```

```
      .toggleClass('hidden');
  });
});
```

Event propagation

The click above responds to a regularly unclickable element. To make this appear clickable, add the following CSS class:

```
.hover {
    cursor: pointer;
    background-color: #afa;
}
```

jQuery provides a hover method that takes two function arguments: the first function will be executed when the mouse cursor enters the selected element, and the second is fired when the cursor leaves.

```
$(() => {
    $('#switcher h3')
      .hover(function() {
        $(this).addClass('hover');
      }, function() {
        $(this).removeClass('hover');
}); });
```

The journey of an event

When an event occurs on a page, an entire hierarchy of DOM elements gets a chance to handle the event. As an example, the div, span, and a could all respond to a click on the anchor below:

```
<div class="foo">
    <span class="bar">
      <a href="http://www.example.com/">
        The quick brown fox jumps over the lazy dog.
</a> </span>

    <p>
      How razorback-jumping frogs can level six piqued gymnasts!
</p> </div>
```

One strategy for allowing multiple elements to respond to a user interaction is called **event**

capturing. With event capturing, the event is first given to the most all-encompassing element, and then to progressively more specific ones. In our example, this means that first the element gets passed the event, then the element, and finally the <a> element.

The opposite strategy is called **event bubbling**. The event gets sent to the most specific element, and after this element has an opportunity to react, the event **bubbles up** to more general elements.

jQuery always registers event handlers for the bubbling phase of the model.

Event bubbling can cause unexpected behavior, especially when the wrong element responds to a mouseover or mouseout event. For example, in the code above, a mouseout on the a will call the mouseout when you leave the anchor and then the span and the div.

Suppose we wish to expand the clickable area that triggers the collapsing or expanding of the style switcher. One way to do this is to move the event handler from the label, <h3>, to its containing element.

```
$(() => {
    $('#switcher')
      .click(() => {
        $('#switcher button').toggleClass('hidden');
}); });
```

The downside is that clicking on a button also collapses the style switcher after the style on the content has been altered. To solve this problem, we need access to the event object. This is a DOM construct that is passed to each element's event handler when it is invoked.

```
$(() => {
    $('#switcher')
      .click(function(event) {
        $('#switcher button').toggleClass('hidden');
}); });
```

Now the event object is available as an event within the handler. In the case of a click event, this will be the actual item clicked on.

```
$(() => {
    $('#switcher')
      .click(function(event) {
        if (event.target == this) {
          $(this)
```

```
            .children('button')
            .toggleClass('hidden');
} });

});
```

The event object provides the .stopPropagation() method, which can halt the bubbling process completely for the event. The complete code showing this method is below:

```
$(() => {
    $('#switcher')
      .click((e) => {
        $(e.currentTarget)
          .children('button')
            .toggleClass('hidden');
    });
});  $(() => {

  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .click((e) => {
      const bodyClass = e.target.id.split('-')[1];
      $('body')
        .removeClass()
        .addClass(bodyClass);
      $(e.target)
        .addClass('selected')
        .removeClass('selected');
      e.stopPropagation();
    });
});
```

e.stopPropagation() is used to prevent any other DOM element from responding to the event.

Preventing default actions

If the click event handler was registered on a link element (<a>) rather than a <button> element outside of a form, there would be another problem. When a user clicks on a link, the browser loads a new page.

If the default action is undesired, calling .stopPropagation() on the event will not help. Instead, the .preventDefault() method stops the event in its tracks before the default action is triggered.

Delegating Events

Event bubbling isn't always a hindrance. One great technique that exploits bubbling is called **event delegation**.

As an example, you can assign a single click handler to an ancestor element in the DOM. An uninterrupted click event will eventually reach the ancestor due to event bubbling, and we can do our work there.

```
$(() => {
    $('#switcher')
      .click((e) => {
        if ($(event.target).is('button')) {
          const bodyClass = e.target.id.split('-')[1];
          $('body')
            .removeClass()
            .addClass(bodyClass);
$(e.target)
  .addClass('selected')
  .removeClass('selected');
       e.stopPropagation();
     }
}); });
```

The .is() method is introduced here, which accepts selector expressions and tests the current jQuery object against the selector. Similarly, you can test for the presence of a class on an element with .hasClass().

However, the code above has an unintentional side-effect. When a button is clicked, the switcher collapses, as it did before we added the call to .stopPropagation(). To Fix:

```
$(() => {
    $('#switcher-default')
      .addClass('selected');
    $('#switcher')
      .click((e) => {
        if ($(e.target).is('button')) {
          const bodyClass = e.target.id.split('-')[1];
          $('body')
            .removeClass()
            .addClass(bodyClass);
          $(e.target)
            .addClass('selected')
            .removeClass('selected');
```

```
    } else {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
}); });
```

Using built-in event delegation capabilities

The .on() method we have already discussed can perform event delegation when provided with appropriate parameters:

```
$(() => {
    $('#switcher-default')
      .addClass('selected');
    $('#switcher')
     .on('click', 'button', (e) => {
       const bodyClass = e.target.id.split('-')[1];
       $('body')
         .removeClass()
         .addClass(bodyClass);
       $(e.target)
         .addClass('selected')
         .siblings('button')
         .removeClass('selected');
       e.stopPropagation();
     })
     .on('click', (e) => {
       $(e.currentTarget)
.children('button')
       .toggleClass('hidden');
   });
});
```

Removing an event handler

There are times when you want an event handler previously registered to be turned off. This is accomplished by calling the .off() method. Code below shows the .off method where when the **Narrow Column** or **Large Print** button is selected, clicking on the background of the style switcher should do nothing

```
$(() => {
  $('#switcher')
  .click((e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
} });

    $('#switcher-narrow, #switcher-large')
      .click(() => {
        $('#switcher').off('click');
      });
});
```

Giving namespaces to event handlers

Need to ensure the .off() call is more specific so that it does not remove both of the click handlers. One way of doing this is to use **event namespacing.** The first parameter passed to .on() is the name of the event to watch for. By using a special syntax after the event, you can subcategorize the event. The .collapse suffix is invisible to the event handling system.

```
$(() => {
  $('#switcher')
    .on('click.collapse', (e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
  } });

    $('#switcher-narrow, #switcher-large')
     .click(() => {
 $('#switcher').off('click.collapse');
     });
});
```

Rebinding events

To reinstate the behavior removed previously, you will need to **rebind** the handler whenever **Default** is clicked. First, we should give our handler function a name so that you can use it more than once without repeating. This shows passing .on() a **function reference** as its second

argument.

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$(e.target).is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
};

  $('#switcher').on('click.collapse', toggleSwitcher);
  $('#switcher-narrow, #switcher-large')
    .click(() => {
      $('#switcher').off('click.collapse');
    });
  $('#switcher-default')
    .click(() => {
      $('#switcher').on('click.collapse', toggleSwitcher);
}); });
```

Simulating user interaction

At times, you may want to execute code bound to an event, even if the event isn't triggered directly by user input. The .trigger() method allows us to do just this. The code below is fired when the page loads, collapsing the switcher

```
$(() => {
  $('#switcher').trigger('click');
});
```

The .trigger() method provides the same set of shortcut methods that .on() does

```
$(() => {
  $('#switcher').click();
});
```

Reacting to keyboard events

There are two types of keyboard events: those that react to the keyboard directly (keyup and keydown) and those that react to text input (keypress). As a safe rule of thumb: if you want to know what key the user pushed, observe the keyup or keydown events.

The target of a keyboard event is the element that currently has the **keyboard focus**. To set up a response to various keypresses, you can create an **object literal** of letters and their corresponding buttons to click.

```
$(() => {
    const triggers = {
      D: 'default',
      N: 'narrow',
      L: 'large'
};
```

Then the functionality:

```
$(document)
  .keyup((e) => {
    const key = String.fromCharCode(e.which);
    if (key in triggers) {
      $(`#switcher-${triggers[key]}`).click();
} }); });
```

Presses of these three keys now simulate mouse clicks on the buttons.

OUR FINISHED CODE FOR THIS CHAPTER:

```
    $(() => {
    // Enable hover effect on the style switcher
    const toggleHover = (e) => {
      $(e.target).toggleClass('hover');
    };
    $('#switcher').hover(toggleHover, toggleHover);
    // Allow the style switcher to expand and collapse.
    const toggleSwitcher = (e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
};

    $('#switcher')
      .on('click', toggleSwitcher)
      // Simulate a click so we start in a collaped state.
      .click();
    // The setBodyClass() function changes the page style.
    // The style switcher state is also updated.
```

```
    const setBodyClass = (className) => {
      $('body')
        .removeClass()
  .addClass(className);
  $('#switcher button').removeClass('selected');
  $(`#switcher-${className}`).addClass('selected');
  $('#switcher').off('click', toggleSwitcher);
  if (className == 'default') {
    $('#switcher').on('click', toggleSwitcher);
} };

// Begin with the switcher-default button "selected"
$('#switcher-default').addClass('selected');
// Map key codes to their corresponding buttons to click
const triggers = {
  D: 'default',
  N: 'narrow',
  L: 'large'
};

// Call setBodyClass() when a button is clicked.
$('#switcher')
  .click((e) => {
    if ($(e.target).is('button')) {
      setBodyClass(e.target.id.split('-')[1]);
    }
});

// Call setBodyClass() when a key is pressed.
$(document)
  .keyup((e) => {
    const key = String.fromCharCode(e.which);
    if (key in triggers) {
      setBodyClass(triggers[key]);
} });

});
```