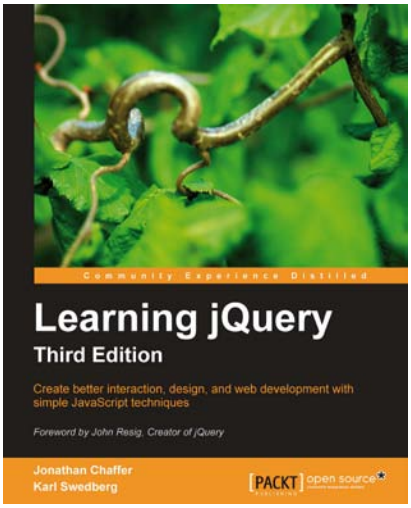
[PACKT] open source
community experience distilled
PUBLISHING

Learning jQuery Third Edition

Jonathan Chaffer
Karl Swedberg



Chapter No.3
"Handling Events"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Handling Events"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Jonathan Chaffer is a member of Rapid Development Group, a web development firm located in Grand Rapids, Michigan. His work there includes overseeing and implementing projects in a wide variety of technologies, with an emphasis in PHP, MySQL, and JavaScript. He also leads on-site training seminars on the jQuery framework for web developers.

In the open-source community, Jonathan has been very active in the Drupal CMS project, which has adopted jQuery as its JavaScript framework of choice. He is the creator of the Content Construction Kit, a popular module for managing structured content on Drupal sites. He is responsible for major overhauls of Drupal's menu system and developer API reference.

Jonathan lives in Grand Rapids with his wife, Jennifer.

I would like to thank Jenny for her tireless enthusiasm and support, Karl for the motivation to continue writing when the spirit is weak, and the Ars Technica community for constant inspiration toward technical excellence. In addition, I'd like to thank Mike Henry and the Twisted Pixel team for producing consistently entertaining distractions in between writing sessions.

Karl Swedberg is a web developer at Fusionary Media in Grand Rapids, Michigan, where he spends much of his time making cool things happen with JavaScript. As a member of the jQuery team, Karl is responsible for maintaining the jQuery API site at `api.jquery.com`. He also publishes tutorials on his blog, `learningjquery.com`, and presents at workshops and conferences. When he isn't coding, Karl likes to hang out with his family, roast coffee in his garage, and exercise at the local cross-fi t gym.

I wish to thank my wife, Sara, and my two children, Benjamin and Lucia, for all the joy that they bring into my life. Thanks also to Jonathan Chaffer for his patience and his willingness to write this book with me.

Many thanks to John Resig for creating the world's greatest JavaScript library and to all the others who have contributed their code, time, and expertise to the project. Thanks to the folks at Packt Publishing, the technical reviewers of this book, the jQuery Cabal, and the many others who have provided help and inspiration along the way.

Learning jQuery Third Edition

In 2005, inspired by pioneers in the field such as Dean Edwards and Simon Willison, John Resig put together a set of functions to make it easy to programmatically find elements on a web page and assign behaviors to them. By the time he first publicly announced his project in January 2006, he had added DOM modification and basic animations. He gave it the name jQuery to emphasize the central role of finding, or querying, parts of a web page and acting on them with JavaScript. In the few short years since then, jQuery has grown in its feature set, improved in its performance, and gained widespread adoption by many of the most popular sites on the Internet. While Resig remains the lead developer of the project, jQuery has blossomed, in true open-source fashion, to the point where it now boasts a core team of top-notch JavaScript developers, as well as a vibrant community of thousands of developers.

The jQuery JavaScript library can enhance your websites regardless of your background. It provides a wide range of features, an easy-to-learn syntax, and robust cross-platform compatibility in a single compact file. What's more, hundreds of plugins have been developed to extend jQuery's functionality, making it an essential tool for nearly every client-side scripting occasion.

Learning jQuery Third Edition provides a gentle introduction to jQuery concepts, allowing you to add interactions and animations to your pages—even if previous attempts at writing JavaScript have left you baffled. This book guides you past the pitfalls associated with Ajax, events, effects, and advanced JavaScript language features, and provides you with a brief reference to the jQuery library to return to again and again.

What This Book Covers

In Chapter 1, Getting Started, you'll get your feet wet with the jQuery JavaScript library. The chapter begins with a description of jQuery and what it can do for you. It then walks you through downloading and setting up the library, as well as writing your first script.

In Chapter 2, Selecting Elements, you'll learn how to use jQuery's selector expressions and DOM traversal methods to find elements on the page, wherever they may be. You'll use jQuery to apply styling to a diverse set of page elements, sometimes in a way that pure CSS cannot.

In Chapter 3, Handling Events, you'll use jQuery's event-handling mechanism to fire off behaviors when browser events occur. You'll see how jQuery makes it easy to attach events to elements unobtrusively, even before the page finishes loading. Also, you'll get an overview of deeper topics, such as event bubbling, delegation, and namespacing.

In Chapter 4, Styling and Animating, you'll be introduced to jQuery's animation techniques and see how to hide, show, and move page elements with effects that are both useful and pleasing to the eye.

In Chapter 5, Manipulating the DOM, you'll learn how to change your page on command. This chapter will teach you how to alter the very structure of an HTML document, as well as its content, on the fly.

In Chapter 6, Sending Data with Ajax, you'll discover the many ways in which jQuery makes it easy to access server-side functionality without resorting to clunky page refreshes. With the basic components of the library well in hand, you will be ready to explore how the library can expand to fit your needs.

In Chapter 7, Using Plugins, will show you how to find, install, and use plugins, including the powerful jQuery UI plugin library.

In Chapter 8, Developing Plugins, you'll learn how to take advantage of jQuery's impressive extension capabilities to develop your own plugins from the ground up. You'll create your own utility functions, add jQuery object methods, and discover the jQuery UI widget factory. Next, you'll take a second tour through jQuery's building blocks, learning more advanced techniques.

In Chapter 9, Advanced Selectors and Traversing, you'll refine your knowledge of selectors and traversals, gaining the ability to optimize selectors for performance, manipulate the DOM element stack, and write plugins that expand selecting and traversing capabilities..

3
Handling Events

JavaScript has several built-in ways of reacting to user interaction and other events. To make a page dynamic and responsive, we need to harness this capability so that we can, at the appropriate times, use the jQuery techniques we have learned so far and the other tricks we'll learn later. While we could do this with vanilla JavaScript, jQuery enhances and extends the basic event handling mechanisms to give them a more elegant syntax, while at the same time making them more powerful.

Performing tasks on page load

We have already seen how to make jQuery react to the loading of a web page. The `$(document).ready()` event handler can be used to fire off a function's worth of code, but there's a bit more to be said about it.

Timing of code execution

In *Chapter 1*, *Getting Started*, we noted that `$(document).ready()` was jQuery's primary way to perform tasks on page load. It is not, however, the only method at our disposal. The native `window.onload` event can achieve a similar effect. While the two methods are similar, it is important to recognize their difference in timing, even though it can be quite subtle depending on the number of resources being loaded.

The `window.onload` event fires when a document is completely downloaded to the browser. This means that every element on the page is ready to be manipulated by JavaScript, which is a boon for writing featureful code without worrying about load order.

On the other hand, a handler registered using `$(document).ready()` is invoked when the DOM is completely ready for use. This also means that all elements are accessible by our scripts, but does not mean that every associated file has been downloaded. As soon as the HTML has been downloaded and parsed into a DOM tree, the code can run.

Style loading and code execution

To ensure that the page has also been styled before the JavaScript code executes, it is a good practice to place `<link rel="stylesheet">` and `<style>` tags prior to `<script>` tags within the document's `<head>` element.

Consider, for example, a page that presents an image gallery; such a page may have many large images on it, which we can hide, show, move, and otherwise manipulate with jQuery. If we set up our interface using the `onload` event, users will have to wait until each and every image is completely downloaded before they can use those features. Even worse, if behaviors are not yet attached to elements that have default behaviors (such as links), user interactions could produce unintended outcomes. However, when we use `$(document).ready()` for the setup, the interface is ready to use earlier with the correct behavior.

What is loaded and what is not?

Using `$(document).ready()` is almost always preferable to using an `onload` handler, but we need to keep in mind that because supporting files may not have loaded, attributes such as image height and width are not necessarily available at this time. If these are needed, we may, at times, also choose to implement an `onload` handler (or more likely, use jQuery to bind a handler to the `load` event); the two mechanisms can coexist peacefully.

Multiple scripts on one page

The traditional mechanism for registering event handlers through JavaScript (rather than adding handler attributes right in HTML) is to assign a function to the DOM element's corresponding attribute. For example, suppose we had defined the function:

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

We could then either assign it within our HTML markup:

```
<body onload="doStuff();">
```

Or, we could assign it from within JavaScript code:

```
window.onload = doStuff;
```

Both of these approaches will cause the function to execute when the page is loaded. The advantage of the second is that the behavior is more cleanly separated from the markup.

Referencing vs. calling functions

Note here, that when we assign a function as a handler, we use the function name but omit the trailing parentheses. With the parentheses, the function is called immediately; without, the name simply identifies, or **references** the function, and can be used to call it later.

With one function, this strategy works quite well. However, suppose we have a second function:

```
function doOtherStuff() {
  // Perform another task...
}
```

We could then attempt to assign this function to run on page load:

```
window.onload = doOtherStuff;
```

However, this assignment trumps the first one. The `.onload` attribute can only store one function reference at a time: so we can't add this to the existing behavior.

The `$(document).ready()` mechanism handles this situation gracefully. Each call to the method adds the new function to an internal queue of behaviors; when the page is loaded all of the functions will execute. The functions will run in the order in which they were registered.

To be fair, jQuery doesn't have a monopoly on workarounds to this issue. We can write a JavaScript function that forms a new function that calls the existing `onload` handler, then calls a passed-in handler. This approach avoids conflicts between rival handlers like `$(document).ready()` does, but lacks some of the other benefits we have discussed. In modern browsers, including Internet Explorer 9, the `DOMContentLoaded` event can be triggered with the W3C standard `document.addEventListener()` method. However, if we need to support older browsers as well, jQuery handles the inconsistencies that these browsers present so that we don't have to.

Shortcuts for code brevity

The `$(document).ready()` construct is actually calling the `.ready()` method on a jQuery object we've constructed from the `document` DOM element. The `$()` function provides a shortcut for us as this is a common task. When we pass in a function as the argument, jQuery performs an implicit call to `.ready()`. For the same result as shown in the following code snippet:

```
$(document).ready(function() {
  // Our code here...
});
```

We can also write the following code:

```
$(function() {
  // Our code here...
});
```

While this other syntax is shorter, the longer version makes code more descriptive about what it is doing. For this reason, we will use the longer syntax throughout this book.

Passing an argument to the .ready() callback

In some cases, it may prove useful to use more than one JavaScript library on the same page. As many libraries make use of the `$` identifier (as it is short and convenient), we need a way to prevent collisions between these uses.

Fortunately, jQuery provides a method called `jQuery.noConflict()` to return control of the `$` identifier back to other libraries. Typical usage of `jQuery.noConflict()` is as follows:

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

First, the other library (Prototype in this example) is included. Then, jQuery itself 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). Now in our custom script, we can use both libraries—but whenever we want to use a jQuery method, we need to write `jQuery` instead of `$` as an identifier.

The `.ready()` method has one more trick up its sleeve to help us in this situation. The callback function we pass to it can take a single parameter: the jQuery object itself. This allows us to effectively rename it without fear of conflicts, as shown in the following code snippet:

```
jQuery(document).ready(function($) {
  // In here, we can use $ like normal!
});
```

Or, using the shorter syntax we learned in the preceding code:

```
jQuery(function($) {
  // Code that uses $.
});
```

Simple events

There are many other times, apart from the loading of the page, at which we might want to perform a task. Just as JavaScript allows us to intercept the page load event with `<body onload="">` or `window.onload`, it provides similar hooks for user-initiated events such as mouse clicks (`onclick`), form fields being modified (`onchange`), and windows changing size (`onresize`). When assigned directly to elements in the DOM, these hooks have similar drawbacks to the ones we outlined for `onload`. Therefore, jQuery offers an improved way of handling these events as well.

A simple style switcher

To illustrate some event handling techniques, suppose we wish to have a single page rendered in several different styles based on user input. We will allow the user to click buttons to toggle between a normal view, a view in which the text is constrained to a narrow column, and a view with large print for the content area.

Progressive enhancement

In a real-world example, a good web citizen will employ the principle of **progressive enhancement** here. The style switcher should either be hidden when JavaScript is unavailable or, better yet, should still function through links to alternative versions of the page. For the purposes of this tutorial, we'll assume that all users have JavaScript turned on.

The HTML markup for the style switcher is as follows:

```
<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>
```

Combined with the rest of the page's HTML markup and some basic CSS, we get a page that looks like the following screenshot:

To begin, we'll make the **Large Print** button operate. We need a bit of CSS to implement our alternative view of the page, as shown in the following code snippet:

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

Our goal, then, is to apply the `large` class to the `<body>` tag. This will allow the stylesheet to reformat the page appropriately. Using what we learned in *Chapter 2*, *Selecting Elements*, the following is the statement needed to accomplish this:

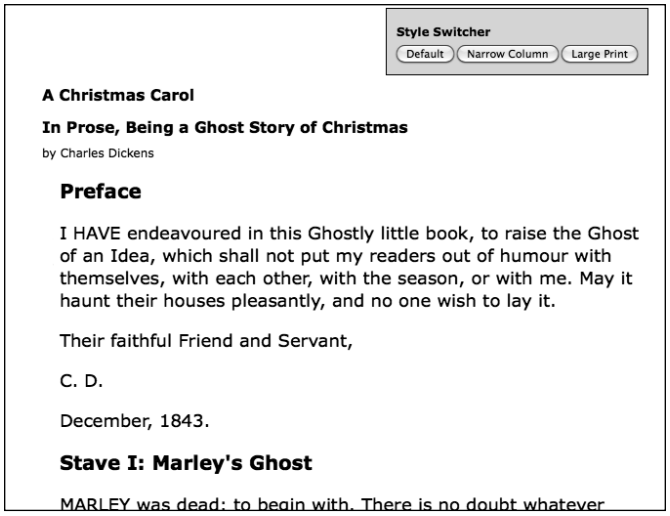
```
$('body').addClass('large');
```

However, we want this to occur when the button is clicked, not when the page is loaded as we have seen so far. To do this, we'll introduce the `.bind()` method. This method allows us to specify any DOM event, and to attach a behavior to it. In this case, the event is called `click`, and the behavior is a function consisting of our preceding one-liner:

```
$(document).ready(function() {
  $('#switcher-large').bind('click', function() {
    $('body').addClass('large');
  });
});
```

Listing 3.1

Now when the button gets clicked, our code runs, and the text is enlarged, as shown in the following screenshot:



That's all there is to binding a behavior to an event. The advantages we discussed with the `.ready()` method apply here, as well. Multiple calls to `.bind()` coexist nicely, appending additional behaviors to the same event as necessary.

This is not necessarily the most elegant or efficient way to accomplish this task. As we proceed through this chapter, we will extend and refine this code into something we can be proud of.

Enabling the other buttons

We now have a **Large Print** button that works as advertised, but we need to apply similar handling to the other two buttons (**Default** and **Narrow Column**) to make them perform their tasks. This is straightforward; we use `.bind()` to add a `click` handler to each of them, removing and adding classes as necessary. The new code reads as shown in the following code snippet:

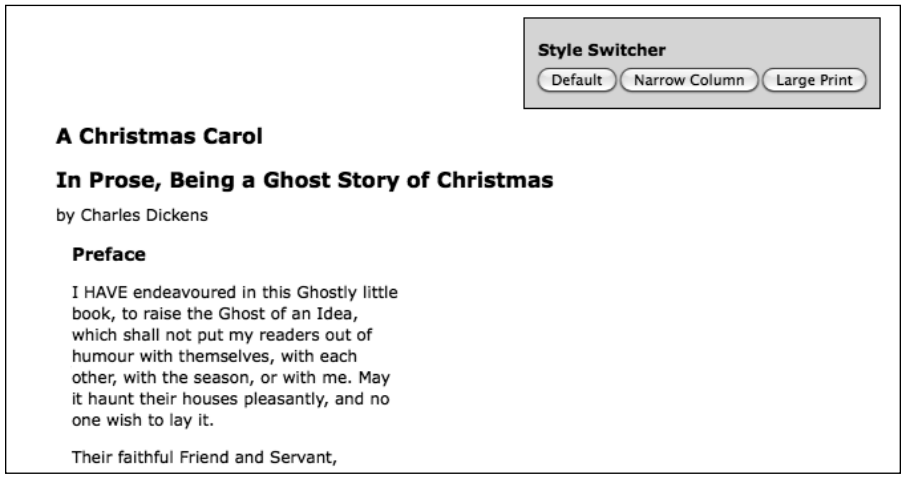
```
$(document).ready(function() {
  $('#switcher-default').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').removeClass('large');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
    $('body').removeClass('large');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').addClass('large');
  });
});
```

Listing 3.2

This is combined with a CSS rule for the `narrow` class:

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

Now, after clicking the **Narrow Column** button, its corresponding CSS is applied and the text gets laid out differently:



Clicking on **Default** removes both class names from the `<body>` tag, returning the page to its initial rendering.

Event handler context

Our switcher is behaving correctly, but we are not giving the user any feedback about which button is currently active. Our approach for handling this will be to apply the `selected` class to the button when it is clicked, and remove this class from the other buttons. The `selected` class simply makes the button's text bold:

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

We could accomplish the preceding class modification by referring to each button by ID and applying or removing classes as necessary, but instead we'll explore a more elegant and scalable solution that exploits the **context** in which event handlers run.

When any event handler is triggered, the keyword `this` refers to the DOM element to which the behavior was attached. Earlier we noted that the `$()` function could take a DOM element as its argument; this is one of the key reasons that facility is available. By writing `$(this)` within the event handler, we create a jQuery object corresponding to the element, and can act on it just as if we had located it with a CSS selector.

With this in mind, we can write the following code snippet:

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

Placing this line in each of the three handlers will add the class when a button is clicked. To remove the class from the other buttons, we can take advantage of jQuery's implicit iteration feature, and write the following code snippet:

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

This line removes the class from every button inside the style switcher.

We should also add the class to the **Default** button when the document is ready. So, placing these in the correct order, we have the following code snippet:

```
$(document).ready(function() {
  $('#switcher-default')
  .addClass('selected')
  .bind('click', function() {
    $('body').removeClass('narrow');
    $('body').removeClass('large');
    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
    $('body').removeClass('large');
    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').addClass('large');
    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Listing 3.3

Now the style switcher gives appropriate feedback.

Generalizing the statements by using the handler context allows us to be yet more efficient. We can factor the highlighting routine out into a separate handler, as shown in Listing 3.4, because it is the same for all three buttons:

```
$(document).ready(function() {
  $('#switcher-default')
  .addClass('selected')
  .bind('click', function() {
    $('body').removeClass('narrow').removeClass('large');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow').removeClass('large');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow').addClass('large');
  });

  $('#switcher button').bind('click', function() {
    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Listing 3.4

This optimization takes advantage of three jQuery features we have discussed. First, **implicit iteration** is, once again, useful when we bind the same `click` handler to each button with a single call to `.bind()`. Second, **behavior queuing** allows us to bind two functions to the same click event, without the second overwriting the first. Lastly, we're using jQuery's **chaining** capabilities to collapse the adding and removing of classes into a single line of code each time.

Further consolidation

The code optimization we've just completed is an example of **refactoring**—modifying existing code to perform the same task in a more efficient or elegant way. To explore further refactoring opportunities, let's look at the behaviors we have bound to each button. The `.removeClass()` method's parameter is optional; when omitted, it removes all classes from the element. We can streamline our code a bit by exploiting this feature, as follows:

```
// work in progress
$(document).ready(function() {
  $('#switcher-default')
```

```
    .addClass('selected')
    .bind('click', function() {
      $('body').removeClass();
    });
    $('#switcher-narrow').bind('click', function() {
      $('body').removeClass().addClass('narrow');
    });
    $('#switcher-large').bind('click', function() {
      $('body').removeClass().addClass('large');
    });

    $('#switcher button').bind('click', function() {
      $('#switcher button').removeClass('selected');
      $(this).addClass('selected');
    });
  });
```

Listing 3.5

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 `.addClass()` we perform in the same breath.

We can only safely remove all classes because we are in charge of the HTML in this case. When we are writing code for reuse (such as for a plugin), we need to respect any classes that might be present and leave them intact.

Now, we are executing some of the same code in each of the buttons' handlers. This can be easily factored out into our general button `click` handler, as shown in the following code snippet:

```
$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher button').bind('click', function() {
    $('body').removeClass();
    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });

  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
  });
```

```
  $('#switcher-large').bind('click', function() {
    $('body').addClass('large');
  });
});
```

Listing 3.6

Note that we need to move the general handler above the specific ones now. The `.removeClass()` needs to happen before the `.addClass()`, and we can count on this because jQuery always triggers event handlers in the order in which they were registered. Finally, we can get rid of the specific handlers entirely by, once again, exploiting **event context**. As the context keyword `this` gives us a DOM element rather than a jQuery object, we can use native DOM properties to determine the ID of the element that was clicked. We can, thus, bind the same handler to all the buttons and within the handler perform different actions for each button, as follows:

```
$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher button').bind('click', function() {
    var bodyClass = this.id.split('-')[1];

    $('body').removeClass().addClass(bodyClass);

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

Listing 3.7

The value of the `bodyClass` variable will be `default`, `narrow`, or `large`, depending on which button is clicked. Here, we are departing somewhat from our previous code in that we are adding a `default` class to the `<body>` when the user clicks `<button id="switcher-default">`. While we do not need this class applied, it isn't causing any harm either, and the reduction of code complexity more than makes up for an unused class name.

Shorthand events

Binding a handler for an event (like 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 `.bind()` counterparts with a few less keystrokes.

For example, our style switcher could be written using `.click()` instead of `.bind()` as shown in the following code snippet:

```
$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher button').click(function() {
    var bodyClass = this.id.split('-')[1];

    $('body').removeClass().addClass(bodyClass);

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

Listing 3.8

Shorthand event methods, such as this, exist for all standard DOM events, as shown in the following list:

- `blur`
- `change`
- `click`
- `dblclick`
- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`

- `scroll`
- `select`
- `submit`
- `unload`

Each shortcut method binds a handler to the event with the corresponding name.

Compound events

Most of jQuery's event-handling methods correspond directly to native DOM events. A handful, however, are custom handlers added for convenience and cross-browser optimization. One of these, the `.ready()` method, we have discussed in detail already. Others, including `.mouseenter()`, `.mouseleave()`, `.focusin()`, and `.focusout()`, normalize proprietary Internet Explorer events of the same name. Two custom jQuery handlers, `.toggle()` and `.hover()`, are referred to as **compound event handlers** because they intercept combinations of user actions, and respond to them using more than one function.

Showing and hiding advanced features

Suppose that we wanted to be able to hide our style switcher when it is not needed. One convenient way to hide advanced features is to make them collapsible. We will allow one click on the label to hide the buttons, leaving the label alone. Another click on the label will restore the buttons. We need another class to handle the hidden buttons, as follows:

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

We could implement this feature by storing the current state of the buttons in a variable, and checking its value each time the label is clicked to know whether to add or remove the hidden class on the buttons. We could also directly check for the presence of the class on a button, and use this information to decide what to do. Instead, jQuery provides the `.toggle()` method, which performs this housekeeping task for us.

Toggle effect

There are in fact two `.toggle()` methods defined by jQuery. For information on the effect method of this name (which is distinguished by different argument types), see: `http://api.jquery.com/toggle/`

The `.toggle()` event method takes two or more arguments, each of which is a function. The first click on the element causes the first function to execute; the second click triggers the second function, and so forth. Once each function has been invoked, the cycle begins again from the first function. With `.toggle()`, we can implement our collapsible style switcher quite easily:

```
$(document).ready(function() {
  $('#switcher h3').toggle(function() {
    $('#switcher button').addClass('hidden');
  }, function() {
    $('#switcher button').removeClass('hidden');
  });
});
```

Listing 3.9

After the first click, the buttons are all hidden:

And a second click returns them to visibility:

Once again, we rely on implicit iteration; this time, to hide all the buttons in one fell swoop without requiring an enclosing element.

For this specific case, jQuery provides another mechanism for the collapsing we are performing. We can use the `.toggleClass()` method to automatically check for the presence of the class before applying or removing it:

```
$(document).ready(function() {
  $('#switcher h3').click(function() {
    $('#switcher button').toggleClass('hidden');
  });
});
```

Listing 3.10

In this case, `.toggleClass()` is probably the more elegant solution, but `.toggle()` is, in general, a versatile way to perform two or more different actions in alternation.

Highlighting clickable items

In illustrating the ability of the `click` event to operate on normally non-clickable page elements, we have crafted an interface that gives few hints that the style switcher label—actually just an `<h3>` element—is actually a live part of the page, awaiting user interaction. To remedy this, we can give it a rollover state, making it clear that it interacts in some way with the mouse:

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

The CSS specification includes a pseudo-class called `:hover`, which allows a stylesheet to affect an element's appearance when the user's mouse cursor hovers over it. In Internet Explorer 6, this capability is restricted to link elements, so we can't use it for other items when we need to support this legacy browser. More importantly, in keeping with **progressive enhancement**, we only want to style the `<h3>` as clickable if we have made it clickable with our jQuery code. Fortunately, jQuery's `.hover()` method allows us to use JavaScript to change an element's styling—and indeed, perform any arbitrary action—both when the mouse cursor enters the element and when it leaves the element.

The `.hover()` method takes two function arguments, just as in our preceding `.toggle()` example. In this case, the first function will be executed when the mouse cursor enters the selected element, and the second is fired when the cursor leaves. We can modify the classes applied to the buttons at these times to achieve a rollover effect, as follows:

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

Listing 3.11

We once again use implicit iteration and event context for short, simple code. Now when hovering over the `<h3>`, we see our class applied in the following screenshot:

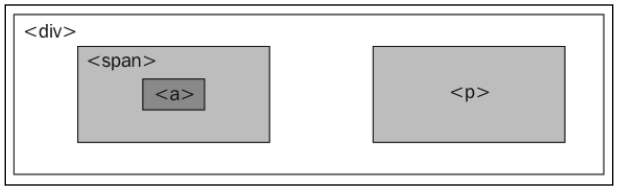
The use of `.hover()` also means we avoid headaches caused by **event propagation** in JavaScript. To understand this, we need to take a look at how JavaScript decides which element gets to handle a given event.

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. Consider a page model similar to the following screenshot:

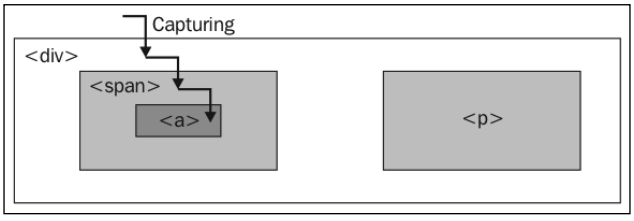
```
<div class="foo">
  <span class="bar">
    <a href="http://www.example.com/">
      The quick brown fox jumps over the lazy dog.
    </a>
  </span>
  <p>
    How razorback-jumping frogs can level six piqued gymnasts!
  </p>
</div>
```

We then visualize the code as a set of nested elements, as shown in the following figure:



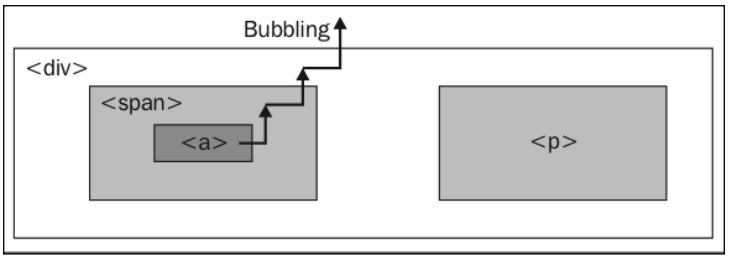
For any event, there are multiple elements that could logically be responsible for reacting. When the link on this page is clicked, for example, the `<div>`, `<span>`, and `<a>` all should get the opportunity to respond to the click. After all, the three are all under the user's mouse cursor at the time. The `<p>` element, on the other hand, is not part of this interaction at all.

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 successively more specific ones. In our example, this means that first the `<div>` gets passed the event, then the `<span>`, and finally, the `<a>`, as illustrated in the following diagram:



Technically, in browser implementations of event capturing, specific elements are registered to listen for events that occur among their descendants. The approximation provided here is close enough for our needs.

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. In our example, the `<a>` would be handed the event first, and then the `<span>` and `<div>` in that order, as illustrated in the following diagram:



Unsurprisingly, different browser developers originally decided on different models for event propagation. The DOM standard that eventually developed thus specified that both strategies should be used: first the event is **captured** from general elements to specific ones, and then the event **bubbles** back up to the top of the DOM tree. Event handlers can be registered for either part of the process.

Not all browsers have been updated to match this new standard, and in those that support capturing it typically must be specifically enabled. To provide cross-browser consistency, therefore, jQuery always registers event handlers for the bubbling phase of the model. We can always assume that the most specific element will get the first opportunity to respond to any event.

Side effects of event bubbling

Event bubbling can cause unexpected behavior, especially when the wrong element responds to a `mouseover` or `mouseout`. Consider a `mouseout` event handler attached to the `<div>` in our example. When the user's mouse cursor exits the `<div>`, the `mouseout` handler is run as anticipated. As this is at the top of the hierarchy, no other elements get the event. On the other hand, when the cursor exits the `<a>` element, a `mouseout` event is sent to that. This event will then bubble up to the `<span>` and then to the `<div>`, firing the same event handler. This bubbling sequence is likely not desired.

The `mouseenter` and `mouseleave` events, either bound individually or combined in the `.hover()` method, are aware of these bubbling issues, and when we use them to attach events, we can ignore the problems caused by the wrong element getting a `mouseover` or `mouseout` event.

The `mouseout` scenario illustrates the need to constrain the scope of an event. While `.hover()` handles this specific case, we will encounter other situations in which we need to limit an event spatially (preventing the event from being sent to certain elements) or temporally (preventing the event from being sent at certain times).

Altering the journey: the event object

We have already seen one situation in which **event bubbling** can cause problems. To show a case in which `.hover()` does not help our cause, we'll alter the collapsing behavior we implemented earlier.

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 `<div>` element:

```
// Unfinished code
$(document).ready(function() {
  $('#switcher').click(function() {
    $('#switcher button').toggleClass('hidden');
  });
});
```

Listing 3.12

This alteration makes the entire area of the style switcher clickable to toggle its visibility. The downside is that clicking on a button also collapses the style switcher after the style on the content has been altered. This is due to event bubbling; the event is first handled by the buttons, then passed up through the DOM tree until it reaches the `<div id="switcher">`, where our new handler is activated and hides the buttons.

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. It provides information about the event, such as where the mouse cursor was at the time of the event. It also provides some methods that can be used to affect the progress of the event through the DOM.

Event object reference

For detailed information about jQuery's implementation of the event object and its properties, see `http://api.jquery.com/category/events/event-object/`

To use the event object in our handlers, we only need to add a parameter to the function, as follows:

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

Note that we have named this parameter `event` because it is descriptive, not because we need to. Naming it `flapjacks` or anything else for that matter would work just as well.

Event targets

Now we have the event object available to us as the variable `event` within our handler. The property `event.target` can be helpful in controlling where an event takes effect. This property is a part of the DOM API, but is not implemented in all browsers; jQuery extends the event object as necessary to provide the property in every browser. With `.target`, we can determine which element in the DOM was the first to receive the event—that is, in the case of a `click` event, the actual item clicked on. Remembering that `this` gives us the DOM element handling the event, we can write the following code:

```
// Unfinished code
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (event.target == this) {
      $('#switcher button').toggleClass('hidden');
    }
  });
});
```

Listing 3.13

This code ensures that the item clicked on was `<div id="switcher">`, not one of its sub-elements. Now clicking on buttons will not collapse the style switcher, and clicking on the switcher's background will. However, clicking on the label, `<h3>`, now does nothing, because it too is a sub-element. Instead of placing this check here, we can modify the behavior of the buttons to achieve our goals.

Stopping event propagation

The event object provides the `.stopPropagation()` method, which can halt the bubbling process completely for the event. Like `.target`, this method is a plain JavaScript feature, but cannot be safely used across all browsers. As long as we register all of our event handlers using jQuery, though, we can use it with impunity.

We'll remove the `event.target == this` check we just added, and instead add some code in our buttons' `click` handlers, as shown in the following code snippet:

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

$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher button').click(function(event) {
    var bodyClass = this.id.split('-')[1];

    $('body').removeClass().addClass(bodyClass);

    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
    event.stopPropagation();
  });
});
```

Listing 3.14

As before, we need to add a parameter to the function we're using as the `click` handler, so we have access to the event object. Then we simply call `event.stopPropagation()` to prevent any other DOM element from responding to the event. Now our click is handled by the buttons, and only the buttons; clicks anywhere else on the style switcher will collapse or expand it.

Default actions

Were our `click` event handler registered on a link element (`<a>`) rather than a generic `<button>` outside of a form, we would face another problem. When a user clicks on a link, the browser loads a new page. This behavior is not an **event handler** in the same sense as the ones we have been discussing; instead, this is the **default action** for a click on a link element. Similarly, when the *Enter* key is pressed while the user is editing a form, the `submit` event is triggered on the form, but then the form submission actually occurs after this.

If these default actions are undesired, then calling `.stopPropagation()` on the event will not help. These actions occur nowhere in the normal flow of event propagation. Instead, the `.preventDefault()` method will serve to stop the event in its tracks before the default action is triggered.

Calling `.preventDefault()` is often useful after we have done some tests on the environment of the event. For example, during a form submission we might wish to check that required fields are filled in, and prevent the default action only if they are not.

Event propagation and default actions are independent mechanisms; either can be stopped while the other still occurs. If we wish to halt both, then we can return `false` at the end of our event handler, which is a shortcut for calling both `.stopPropagation()` and `.preventDefault()` on the event.

Event delegation

Event bubbling isn't always a hindrance; we can often use it to great benefit. One great technique that exploits bubbling is called **event delegation**. With it, we can use an event handler on a single element to do the work of many.

In our example, there are just three `<button>` elements that have attached `click` handlers. However, what if there were many? This is more common than one might think. Consider, for example, a large table of information in which each row has an interactive item requiring a `click` handler. Implicit iteration makes assigning all of these `click` handlers easy, but performance can suffer because of the looping being done internally to jQuery, and because of the memory footprint of maintaining all the handlers.

Instead, we 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.

As an example, let's apply this technique to our style switcher (even though the number of items does not demand the approach). As seen in the preceding Listing 3.13, we can use the `event.target` property to check what element is under the mouse cursor when the click occurs:

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if ($(event.target).is('button')) {
      var bodyClass = event.target.id.split('-')[1];

      $('body').removeClass().addClass(bodyClass);

      $('#switcher button').removeClass('selected');
      $(event.target).addClass('selected');
      event.stopPropagation();
    }
  });
});
```

Listing 3.15

We've used a new method here, called `.is()`. This method accepts the selector expressions we investigated in the previous chapter, and tests the current jQuery object against the selector. If at least one element in the set is matched by the selector, then `.is()` returns `true`. In this case, `$(event.target).is('button')` asks whether the element clicked is a `<button>`. If so, then we proceed with the code from before, with one significant alteration: the keyword `this` now refers to `<div id="switcher">`, so every time we are interested in the clicked button we must now refer to it with `event.target`.

.is() and .hasClass()

We can test for the presence of a class on an element with `.hasClass()`. The `.is()` method is more flexible, however, and can test any selector expression.

We have an unintentional side-effect from this code, however. When a button is clicked now, the switcher collapses, as it did before we added the call to `.stopPropagation()`. The handler for the switcher visibility toggle is now bound to the same element as the handler for the buttons, so halting the event bubbling does not stop the toggle from being triggered. To sidestep this issue, we can remove the `.stopPropagation()` call and instead add another `.is()` test.

Furthermore, as we're making the entire switcher `<div>` clickable, we ought to toggle the class while the user's mouse is over any part of it, as shown in the following code snippet:

```
$(document).ready(function() {
  $('#switcher').hover(function() {
    $(this).addClass('hover');
  }, function() {
    $(this).removeClass('hover');
  });
});

$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  });
});

$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher').click(function(event) {
    if ($(event.target).is('button')) {
      var bodyClass = event.target.id.split('-')[1];

      $('body').removeClass().addClass(bodyClass);

      $('#switcher button').removeClass('selected');
      $(event.target).addClass('selected');
    }
  });
});
```

Listing 3.16

This example is a bit overcomplicated for its size, but as the number of elements with event handlers increases, so does event delegation's benefit. Additionally, we can avoid some of the code repetition by combining the two `click` handlers and using a single if-else statement for the `.is()` test, as follows:

```
$(document).ready(function() {
  $('#switcher-default').addClass('selected');
```

```
  $('#switcher').click(function(event) {
    if ($(event.target).is('button')) {
      var bodyClass = event.target.id.split('-')[1];

      $('body').removeClass().addClass(bodyClass);

      $('#switcher button').removeClass('selected');
      $(event.target).addClass('selected');
    } else {
      $('#switcher button').toggleClass('hidden');
    }
  });
});
```

Listing 3.17

While our code could still use some fine tuning, it is approaching a state at which we can feel comfortable using it for what we set out to do. Nevertheless, for the sake of learning more about jQuery's event handling, we'll back up to Listing 3.16 and continue to modify that version of the code.

Event delegation is also useful in other situations we'll see later, such as when new elements are added by **DOM manipulation** methods (*Chapter 5, Manipulating the DOM*) or **Ajax** routines (*Chapter 6, Sending Data with Ajax*).

Methods for event delegation

As event delegation can be helpful in so many situations, jQuery includes a set of methods specifically for using this technique. We'll fully examine these methods—`.live()`, `.die()`, `.delegate()`, and `.undelegate()`—in *Chapter 10, Advanced Events*. It's worth mentioning now, however, that `.live()` can be used as a drop-in replacement for `.bind()` while providing much of the benefit of event delegation. For example, to bind a live `click` handler to the style-switcher buttons, we can write the following code snippet:

```
$('#switcher button').live('click', function() {
  var bodyClass = event.target.id.split('-')[1];

  $('body').removeClass().addClass(bodyClass);

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

Behind the scenes, jQuery actually binds the `click` handler to the `document` object and uses `event.target` to check if it (or any of its ancestors) matches the selector expression, in this case, `'#switcher button'`. If it does, then jQuery maps the `this` keyword to the matched element. So, while it doesn't provide the potential benefit of avoiding the selection of many elements, it does avoid binding to all of them. Furthermore, because the `document` object itself is available, even when the script is referenced in the `<head>`, `.live()` events can always be bound outside of `$(document).ready()`.

Removing an event handler

There are times when we will be done with an event handler we previously registered. Perhaps the state of the page has changed such that the action no longer makes sense. It is typically possible to handle this situation with conditional statements inside our event handlers, but it may be more elegant to **unbind** the handler entirely.

Suppose that we want our collapsible style switcher to remain expanded whenever the page is not using the normal style. While the **Narrow Column** or **Large Print** button is selected, clicking the background of the style switcher should do nothing. We can accomplish this by calling the `.unbind()` method to remove the collapsing handler when one of the non-default style switcher buttons is clicked:

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  });

  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click');
  });
});
```

Listing 3.18

Now when a button such as **Narrow Column** is clicked, the click handler on the style switcher `<div>` is removed, and clicking the background of the box no longer collapses it. However, the buttons don't work anymore! They are affected by the `click` event of the style switcher `<div>` as well, because we rewrote the button-handling code to use event delegation. This means that when we call `$('#switcher').unbind('click')`, both behaviors are removed.

Event namespacing

We need to make our `.unbind()` call more specific, so that it does not remove both of the click handlers we have registered. One way of doing this is to use **event namespacing**. We can introduce additional information when an event is bound that allows us to identify that particular handler later. To use namespacing, we need to return to the non-shorthand method of binding event handlers, the `.bind()` method itself.

The first parameter we pass to `.bind()` is the name of the event we want to watch for. We can use a special syntax here, though, that allows us to subcategorize the event, as shown in the following code snippet:

```
$(document).ready(function() {
  $('#switcher').bind('click.collapse', function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  });

  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click.collapse');
  });
});
```

Listing 3.19

The `.collapse` suffix is invisible to the event handling system; `click` events are handled by this function, just as if we wrote `.bind('click')`. However, the addition of the namespace means that we can unbind just this handler, without affecting the separate `click` handler we wrote for the buttons.

There are other ways of making our `.unbind()` call more specific, as we will see in a moment. However, event namespacing is a useful tool in our arsenal. It is especially handy in the creation of plugins, as we'll see in later chapters.

Rebinding events

Now clicking the **Narrow Column** or **Large Print** button causes the style switcher collapsing functionality to be disabled. However, we want the behavior to return when the **Default** button is pressed. To do this, we will need to **rebind** the handler whenever **Default** is clicked.

First, we should give our handler function a name so that we can use it more than once without repeating ourselves:

```
$(document).ready(function() {
  var toggleSwitcher = function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  };

  $('#switcher').bind('click.collapse', toggleSwitcher);
});
```

Listing 3.20

Note here, that we are using a new syntax for defining a function. Rather than use a **function declaration** (defining the function by leading with the `function` keyword and naming it), we use an **anonymous function expression**, assigning a no-name function to a **local variable**. Aside from a couple of subtle differences that don't apply in this case, the two syntaxes are functionally equivalent. Here, our use of the function expression is a stylistic choice to make our event handlers and other function definitions resemble each other more closely.

Also, recall that `.bind()` takes a **function reference** as its second argument. It is important to remember when using a named function here to omit parentheses after the function name; parentheses would cause the function to be called, rather than referenced.

Now that the function can be referenced, we can bind it again later without repeating the function definition, as shown in the following code snippet:

```
// Unfinished code
$(document).ready(function() {
  var toggleSwitcher = function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  };

  $('#switcher').bind('click.collapse', toggleSwitcher);

  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click.collapse');
  });
```

```
  $('#switcher-default').click(function() {
    $('#switcher')
      .bind('click.collapse', toggleSwitcher);
  });
});
```

Listing 3.21

Now the toggle behavior is bound when the document is loaded, unbound when **Narrow Column** or **Large Print** is clicked, and rebound when **Normal** is clicked after that.

As we have named the function, we no longer need to use namespacing. The `.unbind()` method can take a function as a second argument; in this case, it unbinds only that specific handler. However, we have run into another problem. Remember that when a handler is bound to an event in jQuery, previous handlers remain in effect. In this case, each time **Normal** is clicked, another copy of the `toggleSwitcher` handler is bound to the style switcher. In other words, the function is called an extra time for each additional click until the user clicks **Narrow** or **Large Print**, which unbinds all of `toggleSwitcher` handlers at once.

When an even number of `toggleSwitcher` handlers are bound, clicks on the style switcher (but not on a button), appear to have no effect. In fact, the `hidden` class is being toggled multiple times, ending up in the same state it was when it began. To remedy this problem, we can unbind the handler when a user clicks on any button, and rebind only after ensuring that the clicked button's ID is `switcher-default`.

```
$(document).ready(function() {
  var toggleSwitcher = function(event) {
    if (!$(event.target).is('button')) {
      $('#switcher button').toggleClass('hidden');
    }
  };

  $('#switcher').bind('click', toggleSwitcher);

  $('#switcher button').click(function() {
    $('#switcher').unbind('click', toggleSwitcher);

    if (this.id == 'switcher-default') {
      $('#switcher').bind('click', toggleSwitcher);
    }

  });
});
```

Listing 3.22

A shortcut is also available for the situation in which we want to unbind an event handler immediately after the first time it is triggered. This shortcut, called `.one()`, is used as follows:

```
$('#switcher').one('click', toggleSwitcher);
```

This would cause the toggle action to occur only once.

Simulating user interaction

At times, it is convenient to execute code that we have bound to an event, even if the normal circumstances of the event are not occurring. For example, suppose we wanted our style switcher to begin in its collapsed state. We could accomplish this by hiding buttons from within the stylesheet, or by adding our `hidden` class or calling the `.hide()` method from a `$(document).ready()` handler. Another way would be to simulate a click on the style switcher so that the toggling mechanism we've already established is triggered.

The `.trigger()` method allows us to do just this:

```
$(document).ready(function() {
  $('#switcher').trigger('click');
});
```

Listing 3.23

Now when the page loads, the switcher is collapsed, just as if it had been clicked, as shown in the following screenshot:

If we were hiding content that we wanted people without JavaScript enabled to see, then this would be a reasonable way to implement **graceful degradation**.

The `.trigger()` method provides the same set of shortcuts that `.bind()` does. When these shortcuts are used with no arguments, the behavior is to trigger the action rather than bind it, as follows:

```
$(document).ready(function() {
  $('#switcher').click();
});
```

Listing 3.24

Keyboard events

As another example, we can add keyboard shortcuts to our style switcher. When the user types the first letter of one of the display styles, we will have the page behave as if the corresponding button were clicked. To implement this feature, we will need to explore **keyboard events**, which behave a bit differently from **mouse 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`). A single character entry event could correspond to several keys: for example, the *Shift* key in combination with the *X* key creates the capital letter **X**. While the specifics of implementation differ from one browser to the next (unsurprisingly), a safe rule of thumb is as follows: if you want to know what key the user pushed, then you should observe the `keyup` or `keydown` event; if you want to know what character ended up on the screen as a result, then you should observe the `keypress` event. For this feature, we just want to know when the user presses the *D*, *N*, or *L* key, so we will use `keyup`.

Next, we need to determine which element should watch for the event. This is a little less obvious than with mouse events, where we have an obvious mouse cursor to tell us about the event's target. Instead, the target of a keyboard event is the element that currently has the **keyboard focus**. The element with focus can be changed in several ways, including mouse clicks and presses of the *Tab* key. Not every element can get the focus either; only items that have default keyboard-driven behaviors such as form fields, links, and elements with a `.tabIndex` property are candidates.

In this case, we don't really care what element has the focus; we want our switcher to work whenever the user presses one of the keys. Event bubbling will once again come in handy, as we can bind our `keyup` event to the `document` element and have assurance that eventually any key event will bubble up to us.

Finally, we will need to know which key was pressed when our `keyup` handler gets triggered. We can inspect the `event` object for this. The `.keyCode` property of the event contains an identifier for the key that was pressed, and for alphabetic keys, this identifier is the ASCII value of the uppercase letter. So we can create a **map**, or **object literal**, of letters and their corresponding buttons to click. When the user presses a key, we'll see if its identifier is in the map, and if so, trigger the click, as follows:

```
$(document).ready(function() {
  var triggers = {
    D: 'default',
    N: 'narrow',
    L: 'large'
  };

  $(document).keyup(function(event) {
    var key = String.fromCharCode(event.keyCode);
    if (key in triggers) {
      $('#switcher-' + triggers[key]).click();
    }
  });
});
```

Listing 3.25

Presses of these three keys now simulate mouse clicks on the buttons—provided that the key event is not interrupted by features such as Firefox's "search for text when I start typing."

As an alternative to using `.trigger()` to simulate this click, let's explore how to factor out code into a function so that more than one handler can call it—in this case, both `click` and `keyup`. While not necessary in this case, this technique can be useful in eliminating code redundancy:

```
$(document).ready(function() {
  // Enable hover effect on the style switcher
  $('#switcher').hover(function() {
    $(this).addClass('hover');
  }, function() {
    $(this).removeClass('hover');
  });

  // Allow the style switcher to expand and collapse
  var toggleSwitcher = function(event) {
```

```
  if (!$(event.target).is('button')) {
    $('#switcher button').toggleClass('hidden');
  }
};
$('#switcher').bind('click', toggleSwitcher);

// Simulate a click so we start in a collapsed state
$('#switcher').click();

// The setBodyClass() function changes the page style
// The style switcher state is also updated
var setBodyClass = function(className) {
  $('body').removeClass().addClass(className);

  $('#switcher button').removeClass('selected');
  $('#switcher-' + className).addClass('selected');

  $('#switcher').unbind('click', toggleSwitcher);

  if (className == 'default') {
    $('#switcher').bind('click', toggleSwitcher);
  }
};

// Begin with the switcher-default button "selected"
$('#switcher-default').addClass('selected');

// Map key codes to their corresponding buttons to click
var triggers = {
  D: 'default',
  N: 'narrow',
  L: 'large'
};

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

// Call setBodyClass() when a key is pressed
```

```
    $(document).keyup(function(event) {
      var key = String.fromCharCode(event.keyCode);
      if (key in triggers) {
        setBodyClass(triggers[key]);
      }
    });
});
```

Listing 3.26

Summary

The abilities we've discussed in this chapter allow us to:

- Use the `.ready()` method to let multiple JavaScript libraries coexist on a single page with `.noConflict()`
- React to a user's click on a page element with **mouse event handlers** and the `.bind()` and `.click()` methods
- Observe **event context** to perform different actions depending on the element clicked, even when the handler is bound to several elements
- Alternately expand and collapse a page element by using `.toggle()`
- Highlight elements under the mouse cursor by using `.hover()`
- Influence **event propagation** and **default actions** to determine which elements get to respond to an event by using `.stopPropagation()` and `.preventDefault()`
- Implement **event delegation** to reduce the number of bound event handlers necessary on a page
- Call `.unbind()` to remove an event handler we're finished with
- Segregate related event handlers with **event namespacing** so they can be acted on as a group
- Cause bound event handlers to execute with `.trigger()`
- Use **keyboard event handlers** to react to a user's key press with `.keyup()`

We can use these capabilities to build quite interactive pages. In the next chapter, we'll learn how to provide visual feedback to the user during these interactions.

Further reading

The topic of event handling will be explored in more detail in *Chapter 10*. A complete list of jQuery's event methods is available in *Appendix C* of this book, in *jQuery Reference Guide*, or in the official jQuery documentation at `http://api.jquery.com/`.

Exercises

To complete these exercises, you will need the `index.html` file for this chapter, as well as the finished JavaScript code as found in `complete.js`. These files can be downloaded from the Packt Publishing website at `http://www.packtpub.com/support`.

Challenge exercises may require use of the official jQuery documentation at `http://api.jquery.com/`.

1. When **Charles Dickens** is clicked, apply the `selected` style to it.
2. When a chapter title (`<h3 class="chapter-title">`) is double-clicked, toggle the visibility of the chapter text.
3. When the user presses the right arrow key, cycle to the next body class. The key code for the right arrow key is `39`.
4. **Challenge**: Use the `console.log()` function to log the coordinates of the mouse as it moves across any paragraph. (Note: `console.log()` displays its results via the Firebug extension for Firefox, Safari's Web Inspector, or the Developer Tools in Chrome).
5. **Challenge**: Use `.mousedown()` and `.mouseup()` to track mouse events anywhere on the page. If the mouse button is released above where it was pressed, then add the `hidden` class to all paragraphs. If it is released below where it was pressed, then remove the `hidden` class from all paragraphs.

For More Information:
www.packtpub.com/learning-jquery-for-interaction-design-web-development-with-javascript/book

Where to buy this book

You can buy Learning jQuery Third Edition from the Packt Publishing website: `http://www.packtpub.com/learning-jquery-for-interaction-design-web-development-with-javascript/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/learning-jquery-for-interaction-design-web-development-with-javascript/book