CHAPTER

5

Action/Reaction: Making Pages Come Alive with Events

When you hear people talk about JavaScript, you usually hear the word "interactive" somewhere in the conversation: "JavaScript lets you make interactive web pages." What they're really saying is that JavaScript lets your web pages react to something a visitor does: moving a mouse over a navigation button produces a menu of links; selecting a radio button reveals a new set of form options; clicking a small photo makes the page darken and a larger version of the photo pop onto the screen.

All the different visitor actions that a web page can respond to are called *events*. JavaScript is an *event-driven* language: Without events, your web pages wouldn't be able to respond to visitors or do anything really interesting. It's like your desktop computer. Once you start it up in the morning, it doesn't do you much good until you start opening programs, clicking files, making menu selections, and moving your mouse around the screen.

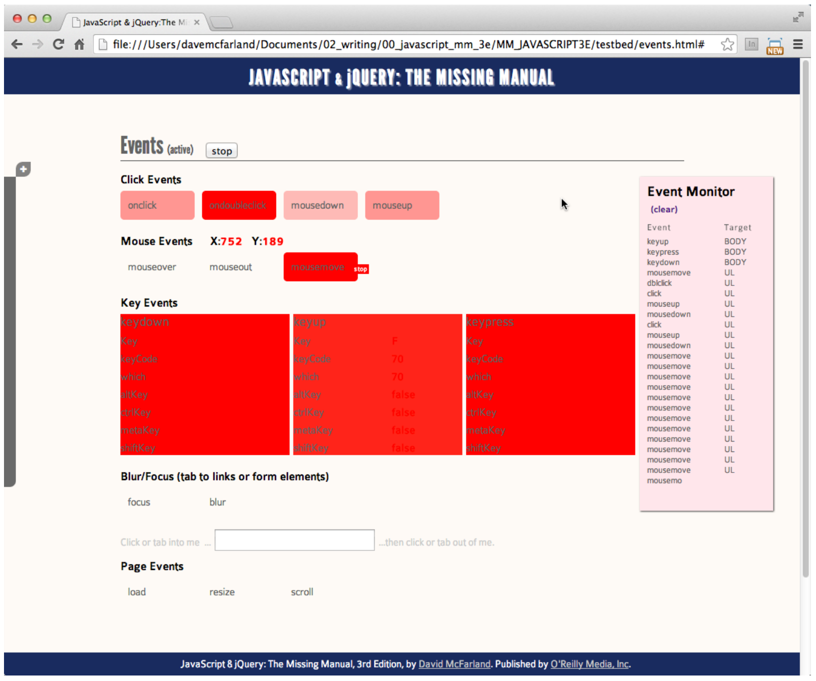
What Are Events?

Web browsers are programmed to recognize basic actions like the page loading, someone moving a mouse, typing a key, or resizing the browser window. Each of the things that happens to a web page is an event. To make your web page interactive, you write programs that respond to events. In this way, you can make a `<div>` tag appear or disappear when a visitor clicks the mouse, a new image appear when she mouses over a link, or check the contents of a text field when she clicks a form's Submit button.

An event represents the precise moment when something happens. For example, when you click a mouse, the precise moment you release the mouse button, the web browser signals that a `click` event has just occurred. The moment that the web browser indicates that an event has happened is when the event *fires*, as programmers put it.

Web browsers actually fire several events whenever you click the mouse button. First, as soon as you press the mouse button, the `mousedown` event fires; then, when you let go of the button, the `mouseup` event fires; and finally, the `click` event fires (Figure 5-1).

NOTE Understanding when and how these events fire can be tricky. To let you test out different event types, this chapter includes a demo web page with the tutorial files. Open *events.html* (in the *testbed* folder) in a web browser. Then move the mouse, click, and type to see some of the many different events that constantly occur on a web page (Figure 5-1).


JavaScript & jQuery:The M
file:///Users/davemcfarland/Documents/02_writing/00_javascript_mm_3e/MM_JAVASCRIPT3E/testbed/events.html#
JAVASCRIPT & jQUERY: THE MISSING MANUAL
Events (active)
stop
Click Events
onclick
ondoubleclick
mousedown
mouseup
Mouse Events
X:752 Y:189
mouseover
mouseout
mousemove
stop
Key Events
Blur/Focus (tab to links or form elements)
focus
blur
Click or tab into me ...
...then click or tab out of me.
Page Events
load
resize
scroll
JavaScript & jQuery: The Missing Manual, 3rd Edition, by David McFarland. Published by O'Reilly Media, Inc.
Event Monitor
(clear)

keydown		keyup		keypress	
Key		Key	F	Key	
keyCode		keyCode	70	keyCode	
which		which	70	which	
altKey		altKey	false	altKey	
ctrlKey		ctrlKey	false	ctrlKey	
metaKey		metaKey	false	metaKey	
shiftKey		shiftKey	false	shiftKey	

```
Event       Target
keyup       BODY
keypress    BODY
keydown     BODY
mousemove   UL
dblclick    UL
click       UL
mouseup     UL
mousedown   UL
click       UL
mouseup     UL
mousedown   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemove   UL
mousemo
```

FIGURE 5-1

While you may not be aware of it, web browsers are constantly firing off events whenever you type, mouse around, or click. For example, double-clicking the mouse triggers two `mousedown`, `mouseup`, *and* `onclick` *events, as well as the doubleclick event itself. The* events.html *file (included with the tutorial files in the* testbed *folder) shows you many of these events in action.*

Mouse Events

Ever since Steve Jobs introduced the Macintosh in 1984, the mouse has been a critical device for all personal computers (as has the trackpad for laptops). Folks use it to open applications, drag files into folders, select items from menus, and even to draw. Naturally, web browsers provide lots of ways of tracking how a visitor uses a mouse to interact with a web page:

- **`click`.** The `click` event fires after you click and release the mouse button. You'll commonly assign a `click` event to a link: For example, a link on a thumbnail image can display a larger version of that image when clicked. However, you're not limited to just links. You can also make any tag on a page respond to an event—even just clicking anywhere on the page.

NOTE The `click` event can also be triggered on links via the keyboard. If you tab to a link, then press the Enter (Return) key, the `click` event fires.

- **`dblclick`.** When you press and release the mouse button twice quickly, a double-click (`dblclick`) event fires. It's the same action you use to open a folder or file on your desktop. Double-clicking a web page isn't a usual web-surfer action, so if you use this event, you should make clear to visitors where they can double-click and what will happen after they do. Also note each click in a double-click event also fires two `click` events, so don't assign `click` and `dblclick` events to the same tag. Otherwise, the function for the click will run twice before the `dblclick` function runs.
- **`mousedown`.** The `mousedown` event is the first half of a click—the moment when you click the button before releasing it. This event is handy for dragging elements around a page. You can let visitors drag items around your web page just like they drag icons around their desktop—by clicking on them (without releasing the button) and moving them, and then releasing the button to drop them (you'll learn how to do this with jQuery UI on page 399).
- **`mouseup`.** The `mouseup` event is the second half of a click—the moment when you release the button. This event is handy for responding to the moment when you drop an item that has been dragged.
- **`mouseover`.** When you move your mouse over an element on a page, a `mouseover` event fires. You can assign an event handler to a navigation button using this event and have a submenu pop up when a visitor mouses over the button. (If you're used to the CSS `:hover` pseudo-class, then you know how this event works.)
- **`mouseout`.** Moving a mouse off an element triggers the `mouseout` event. You can use this event to signal when a visitor has moved her mouse off the page, or to hide a pop-up menu when the mouse travels outside the menu.
- **`mousemove`.** Logically enough, the `mousemove` event fires when the mouse moves—which means this event fires all of the time. You use this event to track the current position of the cursor on the screen. In addition, you can assign this event to a particular tag on the page—a `<div>`, for example—and respond only to movements within that tag.

NOTE Because the `mousemove` event is triggered very frequently (many times as you move the mouse), be careful assigning actions to this event. Trying to process a lengthy action in response to every mouse move can greatly slow down a program and the overall responsiveness of a web page.

Document/Window Events

The browser window itself understands a handful of events that fire from when the page loads to when the visitor leaves the page:

- **`load`.** The `load` event fires when the web browser finishes downloading all of a web page's files: the HTML file itself, plus any linked images, Flash movies, and external CSS and JavaScript files. Web designers have traditionally used this event to start any JavaScript program that manipulated the web page. However, loading a web page and all its files can take a long time if there are a lot of graphics or other large linked files. In some cases, this meant the JavaScript didn't run for quite some time after the page was displayed in the browser. Fortunately, jQuery offers a much more responsive replacement for the `load` event, as described on page 160.
- **`resize`.** When you resize your browser window by clicking the maximize button, or dragging the browser's resize handle, the browser triggers a `resize` event. Some designers use this event to change the layout of the page when a visitor changes the size of his browser. For example, after a visitor resizes his browser window, you can check the window's width—if the window is really wide, you could change the design to add more columns of content to fit the space.
- **`scroll`.** The `scroll` event is triggered whenever you drag the scroll bar, or use the keyboard (for example, the up, down, home, end, and similar keys) or mouse scroll wheel to scroll a web page. If the page doesn't have scrollbars, no `scroll` event is ever triggered. Some programmers use this event to help figure out where elements (after a page has scrolled) appear on the screen.

NOTE Like the `mousemove` event (page 149), the `scroll` event triggers over and over again as a visitor scrolls. So again, be careful assigning any complex actions.

- **`unload`.** When you click a link to go to another page, close a browser tab, or close a browser window, a web browser fires an `unload` event. It's like the last gasp for your JavaScript program and gives you an opportunity to complete one last action before the visitor moves on from your page. Nefarious programmers have used this event to make it very difficult to ever leave a page. Each time a visitor tries to close the page, a new window appears and the page returns. But you can also use this event for good: For example, a program can warn a visitor about a form he's started to fill out but hasn't submitted, or the program could send form data to the web server to save the data before the visitor exits the page.

Form Events

In the pre-JavaScript days, people interacted with websites mainly via clicking links and filling out forms created with HTML. Entering information into a form field was really the only way for visitors to provide input to a website. Because forms are still such an important part of the web, you'll find plenty of form events to play with:

- **`submit`.** Whenever a visitor submits a form, the `submit` event fires. A form might be submitted by clicking the Submit button, or simply by hitting the Enter (Return) key while the cursor is in a text field. You'll most frequently use the `submit` event with form validation—to make sure all required fields are correctly filled out *before* the data is sent to the web server. You'll learn how to validate forms on page 273.
- **`reset`.** Although not as common as they used to be, a Reset button lets you undo any changes you've made to a form. It returns a form to the state it was when the page was loaded. You can run a script when the visitor tries to reset the form by using the `reset` event. For example, if the visitor has made some changes to the form, you might want to pop up a dialog box that asks "Are you sure you want to delete your changes?" The dialog box could give the visitor a chance to click a No button and prevent the process of resetting (erasing) the form.
- **`change`.** Many form fields fire a change event when their status changes: for instance, when someone clicks a radio button, or makes a selection from a drop-down menu. You can use this event to immediately check the selection made in a menu, or which radio button was selected.
- **`focus`.** When you tab or click into a text field, it gives the field focus. In other words, the browser's attention is now focused on that page element. Likewise, selecting a radio button, or clicking a checkbox, gives those elements focus. You can respond to the `focus` event using JavaScript. For example, you could add a helpful instruction inside a text field—"Type your full name." When a visitor clicks in the field (giving it focus), you can erase these instructions, so he has an empty field he can fill out.
- **`blur`.** The `blur` event is the opposite of focus. It's triggered when you exit a currently focused field, by either tabbing or clicking outside the field. The `blur` event is another useful time for form validation. For example, when someone types her email address in a text field, then tabs to the next field, you could immediately check what she's entered to make sure it's a valid email address.

NOTE Focus and blur events also apply to links on a page. When you tab to a link, a `focus` event fires; when you tab (or click) off the link, the `blur` event fires.

Keyboard Events

Web browsers also track when visitors use their keyboards, so you can assign commands to keys or let your visitors control a script by pressing various keys. For example, pressing the space bar could start and stop a JavaScript animation.

Unfortunately, the different browsers handle keyboard events differently, even making it hard to tell which letter was entered! (You'll find one technique for identifying which letter was typed on a keyboard in the Tip on page 165.)

- **`keypress`.** The moment you press a key, the `keypress` event fires. You don't have to let go of the key, either. In fact, the `keypress` event continues to fire, over and over again, as long as you hold the key down, so it's a good way to see if a visitor is holding down the key. For example, if you created a web racecar game you could assign a key to the gas pedal. The player only has to press the key down and hold it down to make the car move.

- **`keydown`.** The keydown event is like the `keypress` event—it's fired when you press a key. Actually, it's fired right *before* the `keypress` event. In Opera, the `keydown` event only fires once. In other browsers, the `keydown` event behaves just like the `keypress` event—it fires over and over as long as the key is pressed.

- **`keyup`.** Finally, the `keyup` event is triggered when you release a key.

Using Events the jQuery Way

Traditionally, programming with events has been tricky. For a long time, Internet Explorer had a completely different way of handling events than other browsers, requiring two sets of code (one for IE and one for all other browsers) to get your code to work. Fortunately, IE9 and later use the same method for handling events as other browsers, so programming is a lot easier. However, there are still a lot of people using IE8, so a good solution that makes programming with events easy and cross-browser compatible is needed. Fortunately, you have jQuery.

As you learned in the last chapter, JavaScript libraries like jQuery solve a lot of the problems with JavaScript programming—including pesky browser incompatibilities. In addition, libraries often simplify basic JavaScript-related tasks. jQuery makes assigning events and *event helpers* (the functions that deal with events) a breeze.

As you saw on page 113, jQuery programming involves (a) selecting a page element and then (b) doing something with that element. In fact, because events are so integral to JavaScript programming, it's better to think of jQuery programming as a three-step process:

1. **Select one or more elements.**

   The previous chapter explained how jQuery lets you use CSS selectors to choose the parts of the page you want to manipulate. When assigning events, you want to select the elements that the visitor will interact with. For example, what will a visitor click—a link, a table cell, an image? If you're assigning a `mouseover` event, what page element does a visitor mouse over to make the action happen?

2. **Assign an event.**

   In jQuery, most DOM events have an equivalent jQuery function. So to assign an event to an element, you just add a period, the event name, and a set of parentheses. So, for example, if you want to add a `mouseover` event to every link on a page, you can do this:

   ```
   $('a').mouseover();
   ```

   To add a click event to an element with an ID of `menu`, you'd write this:

   ```
   $('#menu').click();
   ```

   You can use any of the event names listed on pages page 148–152 (and a couple of jQuery-only events discussed on page 162).

   After adding the event, you still have some work to do. In order for something to happen when the event fires, you must provide a function for the event.

3. **Pass a function to the event.**

   Finally, you need to define what happens when the event fires. To do so, you pass a function to the event. The function contains the commands that will run when the event fires: for example, making a hidden `<div>` tag visible or highlighting a moused-over element.

   You can pass a previously defined function's name like this:

   ```
   $('#start').click(startSlideShow);
   ```

   When you assign a function to an event, you omit the () that you normally add to the end of a function's name to call it. In other words, the following won't work:

   ```
   $('#start').click(startSlideShow());
   ```

   However, the most common way to add a function to an event is to pass an *anonymous function* to the event. You read about anonymous functions on page 138—they're basically a function without a name. The basic structure of an anonymous function looks like this:

   ```
   function() {
   // your code here
   }
   ```

The basic structure for using an anonymous function with an event is pictured in Figure 5-2.

NOTE To learn more about how to work with jQuery and events, visit *http://api.jquery.com/category/events/*.

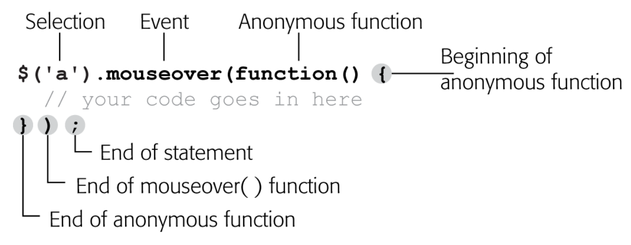

Selection
Event
Anonymous function
$('a').mouseover(function() {
Beginning of anonymous function
// your code goes in here
});
End of statement
End of mouseover( ) function
End of anonymous function

FIGURE 5-2

In jQuery, an event works like a function, so you can pass an argument to the event. You can think of an anonymous function, then, as an argument—like a single piece of data that's passed to a function. If you think of it that way, it's easier to see how all of the little bits of punctuation fit together. For example, in the last line, the } *marks the end of the function (and the end of the argument passed to the* `mouseover` *function); the*) *is the end of the* `mouseover()` *function; and the semicolon is the end of the entire statement that began with the selector* `$('a')`.

Here's a simple example. Assume you have a web page with a link that has an ID of `menu`. When a visitor moves his mouse over that link, you want a hidden list of additional links to appear—assume that the list of links has an ID of `submenu`. So what you want to do is add a `mouseover` event to the menu, and then call a function that shows the submenu. The process breaks down into four steps:

1. **Select the menu:**

   ```
   $('#menu')
   ```

2. **Attach the event:**

   ```
   $('#menu').mouseover();
   ```

3. **Add an anonymous function:**

   ```
   $('#menu').mouseover(function() {

   }); // end mouseover
   ```

   You'll encounter lots of collections of closing brace, closing parenthesis, and semicolons—`});`—which frequently mark the end of an anonymous function inside a function call. You see them everywhere, so it's always a good idea to add a JavaScript comment—in this example, `// end mouseover`—to specify what that trio of punctuation means.

4. **Add the necessary actions (in this case, it's showing the submenu):**

   ```
   $('#menu').mouseover(function() {
    $('#submenu').show();
   }); // end mouseover
   ```

   A lot of people find the crazy nest of punctuation involved with anonymous functions very confusing (that last `});` is always a doozy). And it *is* confusing, but the best way to get used to the strange world of JavaScript is through lots of practice, so the following hands-on tutorial should help reinforce the ideas just presented.

NOTE The `show()` function is discussed in the next chapter on page 184.

Tutorial: Introducing Events

This tutorial gives you a quick introduction to using events. You'll make the page react to several different types of events so you can get a handle on how jQuery events work and how to use them.

NOTE See the note on page 12 for information on how to download the tutorial files.

1. **In a text editor, open the file *events_intro.html* in the *chapter05* folder.**

   You'll start at the beginning by adding a link to the jQuery file.

2. **Click in the empty line just above the closing `</head>` tag and type:**

   ```
   <script src="../_js/jquery.min.js"></script>
   ```

   This line loads the jQuery file from the site. Note that the name of the folder containing the jQuery file is *_js* (don't forget the underscore character at the beginning). Next, you'll add a set of `<script>` tags for your programming.

3. **Press Enter (or Return) to create a new line below the jQuery code and add another set of opening and closing script tags:**

   ```
   <script src="../_js/jquery.min.js"></script>
   <script>

   </script>
   ```

   Now add the `document.ready()` function.

4. **Click in the empty line between the `<script>` tags and add the code in bold:**

   ```
   <script src="../_js/jquery.min.js"></script>
   <script>
   $(document).ready(function() {

   }); // end ready
   </script>
   ```

   Don't forget the JavaScript comment after the `});`. Even though adding comments requires a little extra typing, they'll be very helpful in identifying the different parts of a program. At this point, you've completed the steps you'll follow whenever you use jQuery on your web pages.

Next, it's time to add an event. Your first goal will be simple: have an alert box appear when a visitor double clicks anywhere on the page. To begin, you need to select the element (the page in this case) that you wish to add the event to.

5. **Click in the empty line inside the `.ready()` function and add the bolded code below:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
 $('html')
}); // end ready
</script>
```

The `$('html')` selects the HTML element; basically, the entire browser window. Next, you'll add an event.

6. **Type `.dblclick(); // end double click` after the jQuery selector so your code looks like this:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(); // end double click
}); // end ready
</script>
```

`.dblclick()` is a jQuery function that gets the browser ready to make something happen when a visitor double-clicks on the page. The only thing missing is the "make something happen" part, which requires passing an anonymous function as an argument to the `dblclick()` function (if you need a recap on how functions work and what "passing an argument" means, turn to page 85).

7. **Add an anonymous function by typing the code in bold below:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(function() {

 }); // end double click
}); // end ready
</script>
```

Don't worry, the rest of this book won't crawl through every tutorial at this glacial pace, but it's important for you to understand what each piece of the code is doing. The `function() { }` is just the outer shell; it doesn't do anything until you add programming inside the { and }. That's the next step.

8. **Finally, add an alert statement:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(function() {
    alert('ouch');
  }); // end double click
}); // end ready
</script>
```

If you preview the page in a web browser and double-click anywhere on the page, a JavaScript alert box with the word "ouch" should appear. If it doesn't, double-check your typing to make sure you didn't miss anything.

NOTE After that long build-up, having "ouch" appear on the screen probably feels like a let-down. But keep in mind that the `alert()` part of this script is unimportant—it's all the other code you typed that demonstrates the fundamentals of how to use events with jQuery. As you learn more about programming and jQuery, you can easily replace the alert box with a series of actions that (when a visitor double-clicks the page) moves an element across the screen, displays an interactive photo slideshow, or starts a car-racing game.

Now that you've got the basics, you'll try out a few other events.

1. **Add the code in bold below so your script looks like this:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(function() {
    alert('ouch');
  }); // end double click
 $('a').mouseover(function() {

  }); // end mouseover
}); // end ready
</script>
```

This code selects all links on a page (that's the `$('a')` part), then adds an anonymous function to the `mouseover` event. In other words, when someone mouses over any link on the page, something is going to happen.

2. **Add two JavaScript statements to the anonymous function you added in the last step:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(function() {
    alert('ouch');

  }); // end double click
  $('a').mouseover(function() {
    var message = "<p>You moused over a link</p>";
    $('.main').append(message);
  }); // end mouseover
}); // end ready
</script>
```

The first line here—`var message = "<p>You moused over a link</p>";`—creates a new variable named `message` and stores a string in it. The string represents an HTML paragraph tag with some text. The next line selects an element on the page with a class name of main (that's the `$('.main')`) and then appends (or adds to the end of that element) the contents of the message variable. The page contains a `<div>` tag with the class of main, so this code simply adds "You moused over a link" to the end of that div each time a visitor mouses over a link on the page. (See page 129 for a recap of jQuery's `append()` function.)

3. **Save the page, preview it in a browser, and mouse over any link on the page.**

Each time you mouse over a link, a paragraph is added to the page (Figure 5-3). Now you'll add one last bit of programming: when a visitor clicks on the form button on the page, the browser will change the text that appears on that button.

4. **Lastly, add the code in bold below so your finished script looks like this:**

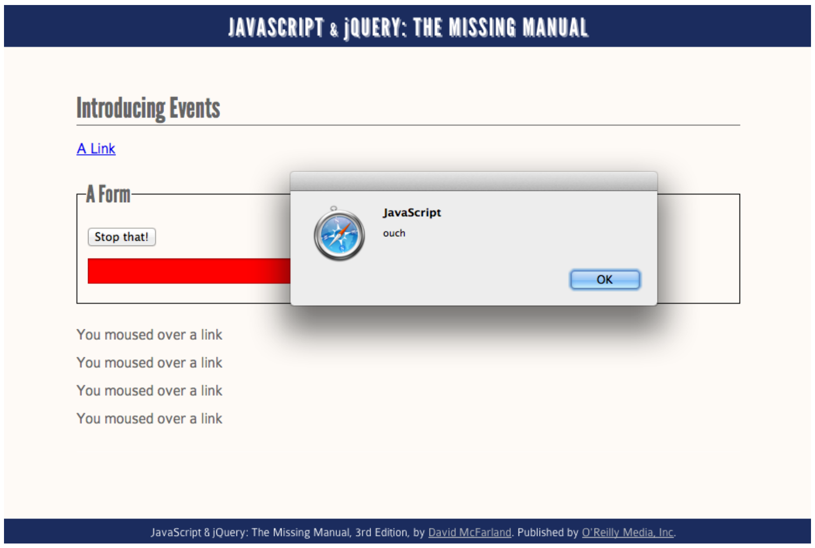
```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('html').dblclick(function() {
    alert('ouch');
  }); // end double click
  $('a').mouseover(function() {
    var message = '<p>You moused over a link</p>';
    $('.main').append(message);
  }); // end mouseover
 $('#button').click(function() {
  $(this).val("Stop that!");
 }); // end click
}); // end ready
</script>
```

You should understand the basics here: `$('#button')` selects an element with the ID button (the form button in this case), and adds a click event to it, so when someone clicks the button, something happens. In this example, the words "Stop that!" appear on the button.

On page 139, you saw how to use `$(this)` inside of a loop in jQuery. It's the same idea inside of an event: `$(this)` refers to the element that is responding to the event—the element you select and attach the event to. In this case, this is the form button. (You'll learn more about the jQuery `val()` function on page 253, but basically you use it to read the value from or change the value of a form element. In this example, passing the string "Stop that!" to the `val()` function sets the button's value to "Stop that!")

5. **Save the page, preview it in a browser, and click the form button.**

   The button's text should instantly change (Figure 5-3). For an added exercise, add the programming to make the text field's background color change to red when a visitor clicks or tabs into it. Here's a hint: You need to (a) select the text field; (b) use the `focus()` event (page 259); (c) use `$(this)` (as in step 12) to address the text field inside the anonymous function; and (d) use the `.css()` function (page 134) to change the background color of the text field. You can find the answer (and a complete version of the page) in the *complete_events_intro.html* file in the *chapter05* folder.



You moused over a link
You moused over a link
You moused over a link
You moused over a link

JavaScript & jQuery: The Missing Manual, 3rd Edition, by David McFarland. Published by O'Reilly Media, Inc.

FIGURE 5-3

jQuery makes it easy for your web pages to respond to user interaction, such as opening an alert box when the page is clicked twice, adding text to the page in response to mousing over a link, or clicking a form button.

More jQuery Event Concepts

Because events are a critical ingredient for adding interactivity to a web page, jQuery includes some special jQuery-only functions that can make your programming easier and your pages more responsive.

Waiting for the HTML to Load

When a page loads, a web browser tries immediately to run any scripts it encounters. So scripts in the head of a page might run before the page fully loads—you saw this in the Moon Quiz tutorial on page 94, where the page was blank until the script asking the questions finished. Unfortunately, this phenomenon often causes problems. Because a lot of JavaScript programming involves manipulating the contents of a web page—displaying a pop-up message when a particular link is clicked, hiding specific page elements, adding stripes to the rows of a table, and so on—you'll end up with JavaScript errors if your program tries to manipulate elements of a page that haven't yet been loaded and displayed by the browser.

The most common way to deal with that problem has been to use a web browser's `onload` event to wait until a page is fully downloaded and displayed before executing any JavaScript. Unfortunately, waiting until a page fully loads before running JavaScript code can create some pretty strange results. The `onload` event only fires *after* all of a web page's files have downloaded—meaning all images, movies, external style sheets, and so on. As a result, on a page with lots of graphics, the visitor might actually be staring at a page for several seconds while the graphics load *before* any JavaScript runs. If the JavaScript makes a lot of changes to the page—for example, styles table rows, hides currently visible menus, or even controls the layout of the page—visitors will suddenly see the page change before their very eyes.

Fortunately, jQuery comes to the rescue. Instead of relying on the load event to trigger a JavaScript function, jQuery has a special function named `ready()` that waits just until the HTML has been loaded into the browser and then runs the page's scripts. That way, the JavaScript can immediately manipulate a web page without having to wait for slow-loading images or movies. (That's actually a complicated and useful feat—another reason to use a JavaScript library.)

You've already used the `ready()` function in a few of the tutorials in this book. The basic structure of the function goes like this:

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

Basically, all of your programming code goes inside this function. In fact, the `ready()` function is so fundamental, you'll probably include it on every page on which you use jQuery. You only need to include it once, and it's usually the first and last line of a script. You must place it within a pair of opening and closing `<script>` tags (it is JavaScript, after all) and after the `<script>` and `</script>` tags that adds jQuery to the page.

So, in the context of a complete web page, the function looks like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Page Title</title>
<script src="js/jquery.js"></script>
<script>
$(document).ready(function() {
    // all of your JavaScript goes in here.
}); // end of ready() function
</script>
</head>
<body>
The web page content...
</body>
</html>
```

TIP Because the `ready()` function is used nearly anytime you add jQuery to a page, there's a shorthand way of writing it. You can remove the `$(document).ready` part, and just type this:

```
$(function() {
  // do something on document ready
});
```

AN ALTERNATIVE TO $(DOCUMENT).READY()

Putting the `$(document).ready()` function in the <head> of an HTML document serves to delay your JavaScript until your HTML loads. But there's another way to do the same thing: put your JavaScript code after the HTML. For example, many web developers simply put their JavaScript code directly before the closing </body> tag, like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Page Title</title>
</head>
<body>
The web page content...
<script src="js/jquery.js"></script>
<script>
    // all of your JavaScript goes in here.
</script>
</body>
</html>
```

In this case, there's no need for `$(document).ready()`, because by the time the scripts load, the document is ready. This approach can have some major benefits. First, you don't need to type that extra bit of code to include the `.ready()` function. Second, loading and running JavaScript freezes the web browser until the scripts load and finish running. If you include a lot of external JavaScript files and they take a while to download, your web page won't display right away. For your site's visitors, it will look like it's taking a long time for your page to load.

You may read on web design blogs that putting your scripts at the bottom of the page is the proper way to add JavaScript. However, there are also downsides to this approach. In some cases, the JavaScript code you add to a page has dramatic effects on the page's appearance. For example, you can use JavaScript to completely re-draw a complex HTML table so it's easier to view and navigate. Or you might take the basic typography of a page and make it look really cool (*http://letteringjs.com*).

In these cases, if you wait until the HTML loads and displays before downloading jQuery and running your JavaScript code, your site's visitors may see the page one way (before it's transformed by JavaScript) and then watch it change right before their eyes. This "quick change act" can be disconcerting. In addition, if you're building a web application that doesn't work without JavaScript, there's no point in showing your visitors the page's HTML, until after the JavaScript loads—after all, the buttons, widgets, and JavaScript-powered interface tools of your web app are just useless chunks of HTML until the JavaScript powers them.

So the answer to where to put your JavaScript is "it depends." In some cases, your site will appear more responsive if you put the JavaScript after the HTML, and sometimes when you put it before. Thankfully, due to browser-caching, once one page on your site downloads the necessary JavaScript files, the other pages will have instant access to the files in the browser cache and won't need to waste time downloading them again. In other words, don't sweat it: if you feel like your web page isn't displaying fast enough, then you can try moving the scripts down to the end of the page. If it helps, then go for it. But, in many cases, whether you use the `.ready()` function at the top of the page won't matter at all.

NOTE When building a web page on your computer and testing it directly in your web browser, you won't encounter the problems discussed in this section. It's only when you put your site on a web server and have to download the script and page files over a sometimes slow Internet connection that you can see whether you have any problems with the time it takes to load and display a web page.

Mousing Over and Off an Element

The `mouseover` and `mouseout` events are frequently used together. For example, when you mouse over a button, a menu might appear; move your mouse off the button, and the menu disappears. Because coupling these two events is so common, jQuery provides a shortcut way of referring to both. jQuery's `hover()` function works just like any other event, except that instead of taking one function as an argument, it accepts two functions. The first function runs when the mouse travels over the element, and the second function runs when the mouse moves off the element. The basic structure looks like this:

```
$('#selector').hover(function1, function2);
```

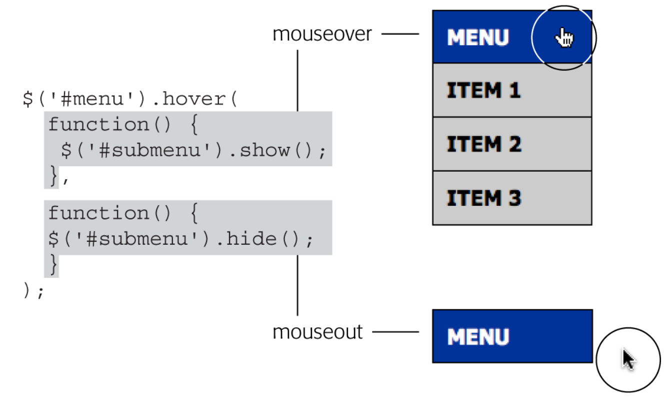
You'll frequently see the `hover()` function used with two anonymous functions. That kind of code can look a little weird; the following example will make it clearer. Suppose when someone mouses over a link with an ID of `menu`, you want a (currently invisible) DIV with an ID of `submenu` to appear. Moving the mouse off of the link hides the submenu again. You can use `hover()` to do that:

```
$('#menu').hover(function() {
   $('#submenu').show();
}, function() {
   $('#submenu').hide();
}); // end hover
```

To make a statement containing multiple anonymous functions easier to read, move each function to its own line. So a slightly more readable version of the code above would look like this:

```
$('#menu').hover(
  function() {
   $('#submenu').show();
  }, // end mouseover
  function() {
   $('#submenu').hide();
  } // end mouseout
); // end hover
```

Figure 5-4 diagrams how this code works for the `mouseover` and `mouseout` events.



```
$('#menu').hover(
  function() {
   $('#submenu').show();
  },
  function() {
  $('#submenu').hide();
  }
);
```

FIGURE 5-4

jQuery's `hover()` *function lets you assign two functions at once. The first function runs when the mouse moves over the element, while the second function runs when the mouse moves off the element.*

If the anonymous function method is just too confusing, you can still use plain old named functions (page 85) to get the job done. First, create a named function to run when the `mouseover` event triggers; create another named function for the `mouseout` event; and finally, pass the names of the two functions to the `hover()` function. In other words, you could rewrite the code above like this:

```
function showSubmenu() {
  $('#submenu').show();
}
function hideSubmenu() {
  $('#submenu').hide();
}
$('#menu').hover(showSubmenu, hideSubmenu);
```

If you find this technique easier, then use it. There's no real difference between the two, though some programmers like the fact that by using anonymous functions you can keep all of the code together in one statement, instead of spread out among several different statements.

NOTE Versions of jQuery prior to 1.9 included a very useful `toggle()` function. This function worked like `hover()` except for click events. You could run one set of code on the first click, and a second set of code on the second click. In other words, you could "toggle" between clicks—great for showing a page item on the first click, then closing that item on the second click. Because `toggle()` is no longer part of jQuery, you'll learn how to replicate that functionality in the tutorial starting on page 174.

The Event Object

Whenever a web browser fires an event, it records information about the event and stores it in an *event object*. The event object contains information that was collected when the event occurred, like the vertical and horizontal coordinates of the mouse, the element on which the event occurred, or whether the Shift key was pressed when the event was triggered.

In jQuery, the event object is available to the function assigned to handling the event. In fact, the object is passed as an argument to the function, so to access it, you just include a parameter name with the function. For example, say you want to find the X and Y position of the cursor when the mouse is clicked anywhere on a page:

```
$(document).click(function(evt) {
  var xPos = evt.pageX;
  var yPos = evt.pageY;
  alert('X:' + xPos + ' Y:' + yPos);
}); // end click
```

The important part here is the `evt` variable. When the function is called (by clicking anywhere in the browser window), the event object is stored in the `evt` variable. Within the body of the function, you can access the different properties of the event object using dot syntax—for example, `evt.pageX` returns the horizontal location of the cursor (in other words, the number of pixels from the left edge of the window).

NOTE In this example, `evt` is just a variable name supplied by the programmer. It's not a special JavaScript keyword, just a variable used to store the event object. You could use any name you want such as `event` or simply `e`.

The event object has many different properties, and (unfortunately) the list of properties varies from browser to browser. Table 5-1 lists some common properties.

TABLE 5-1 *Every event produces an event object with various properties that you can access within the function handling the event*

EVENT PROPERTY	DESCRIPTION
`pageX`	The distance (in pixels) of the mouse pointer from the left edge of the browser window.
`pageY`	The distance (in pixels) of the mouse pointer from the top edge of the browser window.
`screenX`	The distance (in pixels) of the mouse pointer from the left edge of the monitor.
`screenY`	The distance (in pixels) of the mouse pointer from the top edge of the monitor.
`shiftKey`	Is `true` if the shift key is down when the event occurs.
`which`	Use with the `keypress` event to determine the numeric code for the key that was pressed (see tip, next).
`target`	The object that was the "target" of the event—for example, for a click event, the element that was clicked.
`data`	A jQuery object used with the `on()` function to pass data to an event handling function (page 167).

TIP If you access the event object's `which` property with the keypress event, you'll get a numeric code for the key pressed. If you want the specific key that was pressed (a, K, 9, and so on), you need to run the `which` property through a JavaScript method that converts the key number to the actual letter, number, or symbol on the keyboard:

```
String.fromCharCode(evt.which)
```

Stopping an Event's Normal Behavior

Some HTML elements have preprogrammed responses to events. A link, for example, usually loads a new web page when clicked; a form's Submit button sends the form data to a web server for processing when clicked. Sometimes you don't want the web browser to go ahead with its normal behavior. For example, when a form is submitted (the `submit()` event), you might want to stop the form data from being sent if the person filling out the form left out important data.

You can prevent the web browser's normal response to an event with the `preventDefault()` function. This function is actually a part of the event object (see the previous section), so you'll access it within the function handling the event. For example, say a page has a link with an ID of `menu`. The link actually points to another menu page (so visitors with JavaScript turned off will be able to get to the menu page). However, you've added some clever JavaScript, so when a visitor clicks the link, the menu appears right on the same page. Normally, a web browser would follow the link to the menu page, so you need to prevent its default behavior, like this:

```
$('#menu').click(function(evt){
  // clever javascript goes here
 evt.preventDefault(); // don't follow the link
});
```

Another technique is simply to return the value false at the end of the event function. For example, the following is functionally the same as the code above:

```
$('#menu').click(function(evt){
  // clever javascript goes here
  return false; // don't follow the link
});
```

Removing Events

At times, you might want to remove an event that you had previously assigned to a tag. jQuery's `off()` function lets you do just that. To use it, first create a jQuery object with the element you wish to remove the event from. Then add the `off()` function, passing it a string with the event name. For example, if you want to prevent all tags with the class `tabButton` from responding to any `click` events, you can write this:

```
$('.tabButton').off('click');
```

Take a look at a short script to see how the `off()` function works.

```
1   $('a').mouseover(function() {
2     alert('You moved the mouse over me!');
3   });
4   $('#disable').click(function() {
5     $('a').off('mouseover');
6   });
```

Lines 1–3 add a function to the `mouseover` event for all links (`<a>` tags) on the page. Moving the mouse over the link opens an alert box with the message "You moved your mouse over me!" However, because the constant appearance of alert messages would be annoying, lines 4–6 let the visitor turn off the alert. When the visitor clicks a tag with an ID of *disable* (a form button, for example), the `mouseover` events are unbound from all links, and the alert no longer appears.

If you want to remove all events from an element, don't give the `off()` function any arguments. For example, to remove all events—`mouseover`, `click`, `dblclick`, and so on—from a submit button, you could write this code:

```
$('input[type="submit"]').off();
```

This is a pretty heavy-handed approach, however, and in most cases you won't want to remove all event handlers from an element.

NOTE For more information on jQuery's `off()` function, visit *http://api.jquery.com/off/.*

POWER USERS' CLINIC

Stopping an Event in Its Tracks

The event model lets an event pass beyond the element that first receives the event. For example, say you've assigned an event handler for click events on a particular link; when you click the link, the click event fires and a function runs. The event, however, doesn't stop there. Each ancestor element (a tag that wraps around the element that's clicked) can also respond to that same click. So if you've assigned a click event helper for a `<div>` tag that the link is inside, the function for that `<div>` tag's event will run as well.

This concept, known as *event bubbling*, means that more than one element can respond to the same action. Here's another example: Say you add a click event to an image so when the image is clicked, a new graphic replaces it. The image is inside a `<div>` tag to which you've also assigned a click event. In this case, an alert box appears when the `<div>` is clicked. Now when you click the image, both functions will run. In other words, even though you clicked the image, the `<div>` also receives the click event.

You probably won't encounter this situation too frequently, but when you do, the results can be disconcerting. Suppose in the example in the previous paragraph, you don't want the `<div>` to do anything when the image is clicked. In this case, you have to stop the click event from passing on to the `<div>` tag without stopping the event in the function that handles the click event on the image. In other words, when the image is clicked, the function assigned to the image's click event should swap in a new graphic, but then stop the click event.

The `stopPropagation()` function prevents an event from passing onto any ancestor tags. The function is a method of the event object (page 164), so you access it within an event-handling function:

```
$('#theLink').click(function(evt) {
  // do something
  evt.stopPropagation(); // stop event
from continuing
});
```

Advanced Event Management

You can live a long, happy programming life using just the jQuery event methods and concepts discussed on the previous pages. But if you really want to get the most out of jQuery's event-handling techniques, then you'll want to learn about the `on()` function.

NOTE If your head is still aching from the previous section, you can skip ahead to the tutorial on page 174 until you've gained a bit more experience with event handling.

The `on()` method is a more flexible way of dealing with events than jQuery's event-specific functions like `click()` or `mouseover()`. It not only lets you specify an event and a function to respond to the event, but also lets you pass additional data for the event-handling function to use. This lets different elements and events (for example, a click on one link, or a mouseover on an image) pass different information to the same event-handling function—in other words, one function can act differently based on which event is triggered.

NOTE As jQuery has evolved, the names used to add and remove events to elements have changed quite a bit. If you're reading older books or blog posts, you might run into function names like `bind()`, `live()`, and `delegate()`. Those have all been replaced with the `on()` function to add events to elements. In addition, the `off()` function replaces the older `unbind()` function for removing events from elements.

The basic format of the `on()` function is the following:

```
$('#selector').on('click', selector, myData, functionName);
```

The first argument is a string containing the name of the event (like `click`, `mouseover`, or any of the other events listed on pages 148–152).

The second argument is optional, so you don't have to provide a value for this argument when you use the `on()` function. If you do supply the argument, it must be a valid selector like `tr`, `.callout`, or `#alarm`.

NOTE You can use that second argument to apply the event to a different element within the selected element. That technique is called *event delegation,* and you'll learn about it in a bit on page 171.

The third argument is the data you wish to pass to the function—either an object literal or a variable containing an object literal. An object literal (discussed on page 136) is basically a list of property names and values:

```
{
   firstName : 'Bob',
   lastName : 'Smith'
}
```

You can store an object literal in a variable like so:

```
var linkVar = {message:'Hello from a link'};
```

The fourth argument passed to the `on()` function is another function—the one that does something when the event is triggered. The function can either be an anonymous function or named function—in other words, this part is the same as when using a regular jQuery event, as described on page 152.

NOTE Passing data using the `on()` function is optional. If you want to use `on()` merely to attach an event and function, then leave the data variable out:

```
$('selector').on('click', functionName);
```

This code is functionally the same as:

```
$('selector').click(functionName);
```

Suppose you wanted to pop up an alert box in response to an event, but you wanted the message in the alert box to be different based on which element triggered the event. One way to do that would be to create variables with different object literals inside, and then send the variables to the `on()` function for different elements. Here's an example:

```
var linkVar = { message:'Hello from a link'};
var pVar = { message:'Hello from a paragraph'};
function showMessage(evt) {
    alert(evt.data.message);
}
$('a').on('mouseover',linkVar,showMessage);
$('p').on('click',pVar,showMessage);
```

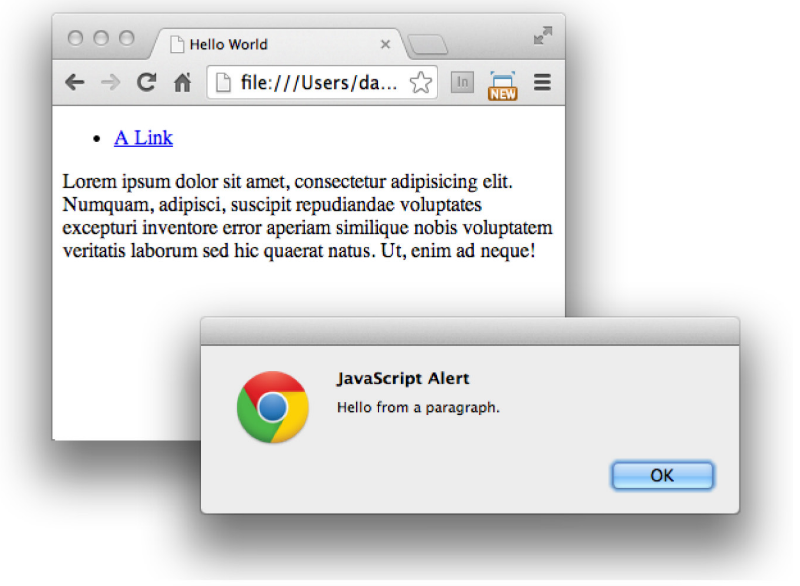
Figure 5-5 breaks down how this code works. It creates two variables, `linkVar` on the first line and `pVar` on the second line. Each variable contains an object literal, with the same property name, `message`, but different message text. A function, `showMessage()`, takes the event object (page 164) and stores it in a variable named `evt`. That function runs the `alert()` command, displaying the `message` property (which is itself a property of the event object's `data` property). Keep in mind that `message` is the name of the property defined in the object literal.

Other Ways to Use the on() Function

jQuery's `on()` function gives you a lot of programming flexibility. In addition to the techniques listed in the previous section, it also lets you tie two or more events to the same function. For example, say you write a program that makes a large image appear on the screen when a visitor clicked a thumbnail image (the common "lightbox" effect found on thousands of websites). You want the larger image to disappear when the visitor either clicks anywhere on the page or hits any key on the keyboard (providing both options makes your program respond to people who prefer the keyboard over the mouse and vice versa). Here's some code that does that:

```
$(document).on('click keypress', function() {
  $('#lightbox').hide();
}); // end on
```

The important part is `'click keypress'`. By providing multiple event names, each separated by a space, you're telling jQuery to run the anonymous function when *any* of the events in the list happen. In this case, when either the click or keypress event fires on the document.

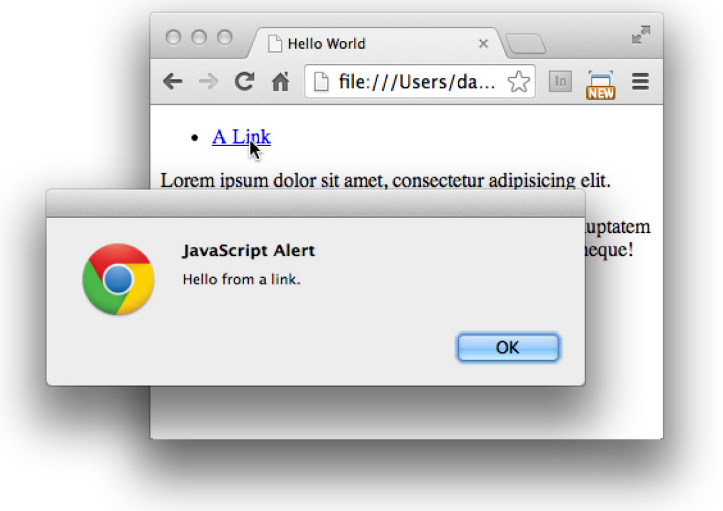


- A Link

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Numquam, adipisci, suscipit repudiandae voluptates excepturi inventore error aperiam similique nobis voluptatem veritatis laborum sed hic quaerat natus. Ut, enim ad neque!

JavaScript Alert

Hello from a paragraph.



- A Link

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

JavaScript Alert

Hello from a link.

FIGURE 5-5

jQuery's `on()` *function lets you pass data to the function responding to the event. That way, you can use a single named function for several different elements (even with different types of events), while letting the function use data specific to each event helper.*

In addition, if you want to attach several events that each trigger different actions, you don't need to use the on() function multiple times. In other words, if you want to make one thing happen when a visitor clicks an element, and another when a visitor mouses over that same element, you might be tempted to write this:

```
$('#theElement').on('click', function() {
  // do something interesting
}); // end on
$('#theElement').on('mouseover', function() {
  // do something else interesting
}); // end on
```

You can do the same thing by passing an object literal (page 136) to the on() function that is composed of an event name, followed by a colon, followed by an anonymous function. Here's the code above rewritten, calling the on() function only once and passing it an object literal (in bold):

```
$('#theElement').on({
  'click' : function() {
    // do something interesting
   }, // end click function
   'mouseover' : function() {
     // do something interesting
   } // end mouseover function
}); // end on
```

Delegating Events with on()

As mentioned on page 167, the on() method can accept a second argument, which is another selector:

```
$('#selector').on('click', selector, myData, functionName);
```

That second argument can be any valid jQuery selector, like an ID, class, element, or any of the selectors discussed on page 119. Passing a selector to the on() function significantly changes how on() works. Without passing a selector, the event is applied to the initial selector—$('#selector')—in the example above. Say you added this code to a page:

```
$('li').on('click', function() {
  $(this).css('text-decoration': 'line-through');
}); // end on
```

This code adds a line through the text in every `<li>` tag that a visitor clicks. Remember that, in this case, $(this) refers to the element that's handling the event—the clicked `<li>` tag. In other words, the click event is "bound" to the `<li>` tag. In most cases, that's exactly what you want to do—define a function that runs when the visitor interacts with a specific element. You can have the function run when a visitor clicks a link, mouses over a menu item, submits a form, and so on.

NOTE Still not sure what `$(this)` stands for? See page 139 for an in-depth discussion, and page 159 for details on how `$(this)` works with event handlers.

However, there's one problem with this method of attaching event handlers to elements: it only works if the element is already on the page. If you dynamically add HTML after you add an event handler like `click()`, `mouseover()`, or `on()`, those new elements won't have any event handlers attached to them. OK, that's a mindful. Here's an example to make it clearer.

Imagine you've created a To-Do List application that lets a visitor manage a list of tasks. When the application first loads, there's nothing in the list (#1 in Figure 5-6). A visitor can fill out a text field and click an Add Task button to add more tasks to the list (#2 in Figure 5-6). After the visitor finishes a task, she can click that task to mark it done (#3 in Figure 5-6).

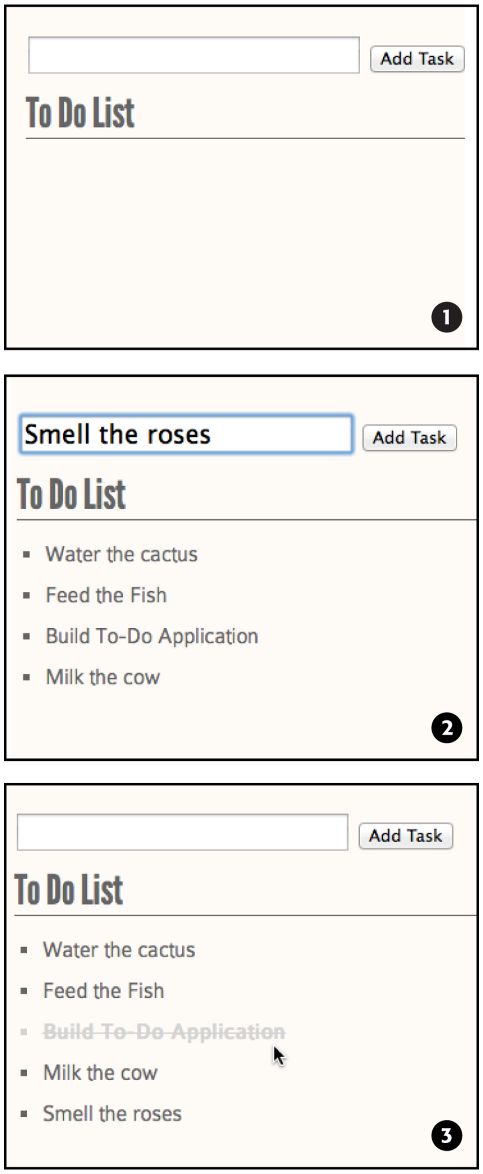
To mark a task done, you know you need to add a click event to each `<li>` tag so when it's clicked, the `<li>` is marked as done in some way. In Figure 5-6, a completed task is grayed-out and has a line through it. The problem is that when the page loads there are no `<li>` tags, so adding a click event handler to every `<li>` tag won't have any effect. In other words, the code on page 171 won't work.

NOTE Note: You can learn more about jQuery event delegation at *http://api.jquery.com/on/.*

Instead, you have to *delegate* the events, which means applying an event to a parent element higher up in the chain (an element that already exists on the page) and then listening for events on particular child elements. Because the event is applied to an already existing element, adding new children won't interfere with this process. In other words, you're *delegating* the task of listening to events to an already existing parent element. For a more detailed explanation, see the box on page 175. Meanwhile, here's how you can use the `on()` function to make this particular example work:

```
$('ul').on('click', 'li', function() {
  $(this).css('text-decoration': 'line-through');
}); // end on
```

When you created the page, you added an empty `<ul>` tag as a container for adding each new task inside an `<li>` tag. As a result, when the page loads, one empty `<ul>` tag is already in place. Then, running the above code adds the `on()` function to that tag. The visitor hasn't yet added any to-do list items, so there are no `<li>` tags. However, when you add the selector `'li'` as the second argument in the `on()` function, you're saying that you want to listen to click events *not* on the `<ul>` tag but on any `<li>` tags inside that unordered list. It doesn't matter when the `<li>` tags are added to the page because the `<ul>` tag is the one doing the event listening.



- Water the cactus
- Feed the Fish
- Build To-Do Application
- Milk the cow

- Water the cactus
- Feed the Fish
- ~~Build To-Do Application~~
- Milk the cow
- Smell the roses

FIGURE 5-6

Sometimes you'll write JavaScript code that dynamically adds new HTML to a page. In this example, visitors can add new items (tasks, in this example) to an unordered list. When the page first loads there are no items in the list, just an empty pair of `<ul> </ul>` *tags (top). As a visitor types in new tasks and clicks the Add Task button, new* `<li>` *tags are added to the page (middle). To mark a task done, just click the task in the list (bottom). You'll find this working example in the tutorial file* to-do-list.html *in the* chapter05 *folder.*

HOW EVENT DELEGATION AFFECTS THE $(THIS) OBJECT

As mentioned earlier, the `$(this)` object refers to the element that's currently being handled within a loop (page 139) or by an event handler (page 159). Normally, in an event handler, `$(this)` refers to the initial selector. For example:

```
$('ul').on('click', function() {
  $(this).css('text-decoration': 'line-through');
}
```

In the above code, `$(this)` refers to the `<ul>` tag the visitor clicks. However, when you use event delegation, the initial selector is no longer the element that's being interacted with—it's the element that contains the element the visitor clicks on (or mouses over, tabs to, and so on). Take a look at the event delegation code one more time:

```
$('ul').on('click', 'li', function() {
  $(this).css('text-decoration': 'line-through');
}); // end on
```

In this case, the `<li>` tag is the one the visitor will click, and it's the element that needs to respond to the event. In other words, `<ul>` is just the container, and the function needs to be run when the `<li>` tag is clicked. So, in this example, `$(this)` is going to refer to the `<li>` tag, and the function above will add a line through each `<li>` tag the visitor clicks.

For many tasks, you won't need event delegation at all. However, if you ever need to add events to HTML that isn't already on the page when it loads, then you'll need to use this technique. For example, when you use Ajax (Chapter 13), you may need to use event delegation to apply events to HTML content that's dynamically added to a web page from a web server.

NOTE In some cases, you may want to use event delegation simply to improve JavaScript performance. If you're adding lots and lots of tags to the same event handler, for example, hundreds of table cells in a large table, it's often better to delegate the event to the `<table>` tag like this:

```
$('table').on('click', 'td', function () {
  // code goes here
});
```

By adding the event to the table, you're avoiding having to apply event handlers directly to hundreds or even thousands of individual elements, a task that can consume browser memory and processing power.

Tutorial: A One-Page FAQ

"Frequently Asked Questions" pages are a common sight on the Web. They can help improve customer service by providing immediate answers 24/7. Unfortunately, most FAQ pages are either one very long page full of questions and complete answers, or a single page of questions that link to separate answer pages. Both solutions slow down the visitors' quest for answers: in the first case, forcing a visitor to scroll down a long page for the question and answer he's after, and in the second case, making the visitor wait for a new page to load.

In this tutorial, you'll solve this problem by creating a JavaScript-driven FAQ page. All of the questions will be visible when the page loads, so it's easy to locate a given question. The answers, however, are hidden until the question is clicked—then the desired answer fades smoothly into view (Figure 5-7).

Overview of the Task

The JavaScript for this task will need to accomplish several things:

- When a question is clicked, the corresponding answer will appear.
- When a question whose answer is visible is clicked, then the answer should disappear.

FREQUENTLY ASKED QUESTION

Is Event Delegation Voodoo?

OK, I understand the basics of event delegation, but how does it actually work?

As you read in the box on page 167, event "bubbles" up through the HTML of a page. When you click on a link inside a paragraph, the `click` event is first triggered on that link; then, the parent element—the paragraph—gets the `click` event, followed by the `<body>` and then the `<html>`. In other words, an event that's triggered on one HTML element passes upwards to each of its parents.

This fact can be really helpful in the case of event delegation. As mentioned on page 171, sometimes you want to apply events to HTML that doesn't yet exist—like an item in a to-do list—that only comes into existence after the page loads and a visitor adds a list item. Although you can't add a `click` event to a `<li>` tag that isn't there yet, you can add the `click` event to an already existing parent like a `<ul>` tag, or even a `<div>` tag that holds that `<ul>` tag.

As discussed on page 164, every event has an event object that keeps track of lots of different pieces of information. In this case, the most important piece of information is the `target` property. This property specifies the exact HTML tag receiving the event. For example, when you click a link, that link is the target of the click. Because of event bubbling, a parent tag can "hear" the event and then determine which child element was the target.

So, with event delegation, you can have a parent element listen for events—like a `<ul>` tag listening for a `click` event. Then, it can check to see which tag was the actual target. For example, if the `<li>` tag was the target, then you can run some specific code to handle that situation—like crossing off the list item as in the to-do list example.

In addition, you'll want to use JavaScript to hide all of the answers when the page loads. Why not just use CSS to hide the answers to begin with? For example, setting the CSS `display` property to none for the answers is another way to hide the answers. The problem with this technique is what happens to visitors who don't have JavaScript turned on: They won't see the answers when the page loads, nor will they be able to make them visible by clicking the questions. To make your pages viewable to both those with JavaScript enabled and those with JavaScript turned off, it's best to use JavaScript to hide any page content.

NOTE See the note on page 12 for information on how to download the tutorial files.

The Programming

1. **In a text editor, open the file *faq.html* in the *chapter05* folder.**

   This file already contains a link to the jQuery file, and the `$(document).ready()` function (page 160) is in place. First, you'll hide all of the answers when the page loads.

2. **Click in the empty line after the `$(document).ready()` function, and then type `$('.answer').hide();`.**

   The text of each answer is contained within a `<div>` tag with the class of `answer`. This one line of code selects each `<div>` and hides it (the `hide()` function is discussed on page 184). Save the page and open it in a web browser. The answers should all be hidden.

NOTE The elements are hidden with JavaScript instead of CSS, because some visitors may not have JavaScript turned on. If that's the case, they won't get the cool effect you're programming in this tutorial, but they'll at least be able to see all of the answers.

   The next step is determining which elements you need to add an event listener to. The answer appears when a visitor clicks the question, therefore you must select every question in the FAQ. On this page, each question is a `<h2>` tag in the page's main body.

3. **Press Return to create a new line and add the code in bold below to the script:**

```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2')
}); // end of ready()
</script>
```

   That's a basic descendant selector used to target every `<h2>` tag inside an element with a class of `main` (so it doesn't affect any `<h2>` tags elsewhere on the page). Now it's time to add an event handler. The `click` event is a good candidate, but you'll need to do something more to make it work. For this example, each click will either show the answer or hide it. That sounds like a job for a conditional statement. Essentially, you want to check whether the answer `<div>` that's placed after the `<h2>` tag is hidden: if it is, then show it, if not, then hide it.

4. **Add the following bolded code to attach an event handler to the `<h2>` tags:**

```
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2').click(function() {

 }); // end click
}); // end of ready()
```

This code adds a `click` event to those `<h2>` tags. The `// end click` comment isn't required, but, as mentioned on page 58, a comment can really help you figure out what code a set of `});` characters belongs to. With this code in place, whatever you put inside the anonymous function here will run each time a visitor clicks one of those `<h2>` tags.

5. **On the empty line inside the function, type:**

```
var $answer = $(this).next('.answer');
```

Here, you're creating a variable—`$answer`—that will hold a jQuery object. As discussed on page 159, `$(this)` refers to the element currently responding to the event—in this case, a particular `<h2>` tag. jQuery provides several functions to make moving around a page's structure easier. The `.next()` function finds the tag immediately following the current tag. In other words, it finds the tag following the `<h2>` tag. You can further refine this search by passing an additional selector to the `.next()` function—the code `.next('.answer')` finds the first tag following the `<h2>` that also has the class `answer`.

In other words, you're storing a reference to the `<div>` immediately following the `<h2>` tag, That `<div>` holds the answer to the question. You're storing it in a variable, because you'll need to access that element several times in the function: to see whether the answer is hidden, to show the answer if it is hidden, and to hide it if it's visible. Every time you access jQuery using the `$( )`, you're telling the browser to fire off a bunch of programming code inside jQuery. So the code `$('this').next('.answer')` makes jQuery do some work. Instead of repeating those same steps over and over, you can just store the results of that work in a variable, and use that variable again and again to point to the `<div>` you wish to hide or show.

When you need to use the same jQuery results over and over and over again, it's a good idea to store the results a single time in a variable you can use repeatedly. This makes the program more efficient, saves the browser from having to unnecessarily do extra work, and make your web page more responsive.

The variable begins with a `$` symbol (as in `$answer`) to show that you're storing a jQuery object (the result of running the `$( )` function). It's not a requirement that you add the `$` in front of the variable name; it's just a common convention jQuery programmers use to let them know that they can use all the wonderful jQuery functions—like `.hide()`—with that variable.

NOTE The `.next()` function is just one of the many jQuery functions (also called *methods*) that help you navigate through a page's DOM. To learn about other helpful functions, visit *http://docs.jquery.com/Traversing.* You can also read more in "Traversing the DOM" on page 531.

6. **Add an empty if/else clause to your code, like this:**

```
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2').click(function() {
  var $answer = $(this).next('.answer');
  if ( ) {

  } else {

  }
 }); // end click
}); // end of ready()
```

Experienced programmers don't usually type an empty if/else statement like this, but as you're learning it can be really helpful to build up your code one piece at a time. Now, why not test to see whether the answer is currently hidden.

7. **Type `$(answer.is(':hidden')` inside the parentheses of the conditional statement:**

```
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2').click(function() {
  var $answer = $(this).next('.answer');
  if ($answer.is(':hidden')) {

  } else {

  }
 }); // end click
}); // end of ready()
```

You're using the `$answer` variable you created in step 5. That variable contains the element with the class `answer` that appears immediately after the `<h2>` tag the visitor clicks. Remember, that's a `<div>` containing the answer to the question inside the `<h2>` tag.

You're using jQuery's `is()` method to see whether that particular element is hidden. The `is()` method checks to see whether the current element matches a particular selector. You give the function any CSS or jQuery selector, and that function tests whether the object matches the given selector. If it does, the function returns a `true` value; if not, it returns a `false` value. That's perfect! As you know, a conditional statement needs either a `true` or `false` value to work (page 66).

The :hidden selector used here is a special, jQuery-only selector that identifies hidden elements. In this case, you're checking to see whether the answer is currently hidden. If not, then you can show it.

8. **Add line 6 below to your program:**

```
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2').click(function() {
  var $answer = $(this).next('.answer');
  if ($answer.is(':hidden')) {
   $answer.slideDown();
  } else {

  }
 }); // end click
}); // end of ready()
```

The slideDown() function is one of jQuery's animation functions (you'll learn more about animation in the next chapter). It reveals a hidden element by sliding it down onto the page. At this point, you can check out your hard work. Save the page and check it out in a web browser. Click one of the questions on the page. The answer below it should open (if it doesn't, double-check your typing and refer to the troubleshooting tips on page 18).

If you look at the page, you'll see a blue + symbol to the left of each headline. The plus sign is a common icon used to mean, "Hey, there's more here." To indicate that a visitor can click to hide the answer, replace the plus sign with a minus sign. You can do it easily by adding a class to the <h2> tag.

9. **After the line of code you added in the last step, type the following:**

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

Remember that $(this) applies to the element that's receiving the event (page 159). In this case, that's the <h2> tag. Thus, this new line of code adds a class named close when the answer is shown. The minus sign icon is defined within the style sheet as a background image. (Once again, CSS makes JavaScript programming easier.)

In the next step, you'll complete the second half of the toggling effect—hiding the answer when the question is clicked a second time.

10. **In the `else` section of the conditional statement, add two more lines of code (bolded below). The finished code should look like this:**

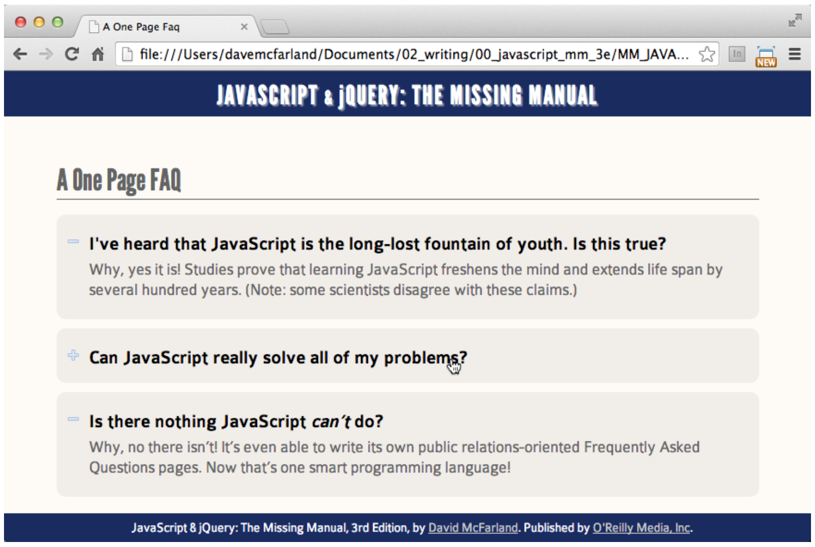
```
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('.answer').hide();
 $('.main h2').click(function() {
  var $answer = $(this).next('.answer');
  if ($answer.is(':hidden')) {
   $answer.slideDown();
   $(this).addClass('close');
  } else {
   $answer.fadeOut();
   $(this).removeClass('close');
  }
 }); // end click
}); // end of ready()
</script>
```

This part of your program hides the answer. You could have used the `slideUp()` function, which hides the element by sliding it up and out of view, but to add interest and variation, in this case you'll fade the answer out of view using the `fadeOut()` function (about which you'll learn more on page 185).

Finally, you'll remove the `close` class from the `<h2>` tag: this makes the + sign reappear at the left of the headline.

Save the page and try it out. Now when you click a question, not only does the answer appear, but the question icon changes (Figure 5-7).

TIP Once you're done, try replacing the `slideDown()` function with `fadeIn()` and the `fadeOut()` function with `slideUp()`. Which of these animation functions do you prefer?


A One Page Faq
file:///Users/davemcfarland/Documents/02_writing/00_javascript_mm_3e/MM_JAVA...

JAVASCRIPT & jQUERY: THE MISSING MANUAL

A One Page FAQ

− I've heard that JavaScript is the long-lost fountain of youth. Is this true?

Why, yes it is! Studies prove that learning JavaScript freshens the mind and extends life span by several hundred years. (Note: some scientists disagree with these claims.)

+ Can JavaScript really solve all of my problems?

− Is there nothing JavaScript *can't* do?

Why, no there isn't! It's even able to write its own public relations-oriented Frequently Asked Questions pages. Now that's one smart programming language!

JavaScript & jQuery: The Missing Manual, 3rd Edition, by David McFarland. Published by O'Reilly Media, Inc.

FIGURE 5-7

With just a few lines of JavaScript, you can make page elements appear or disappear with a click of the mouse.