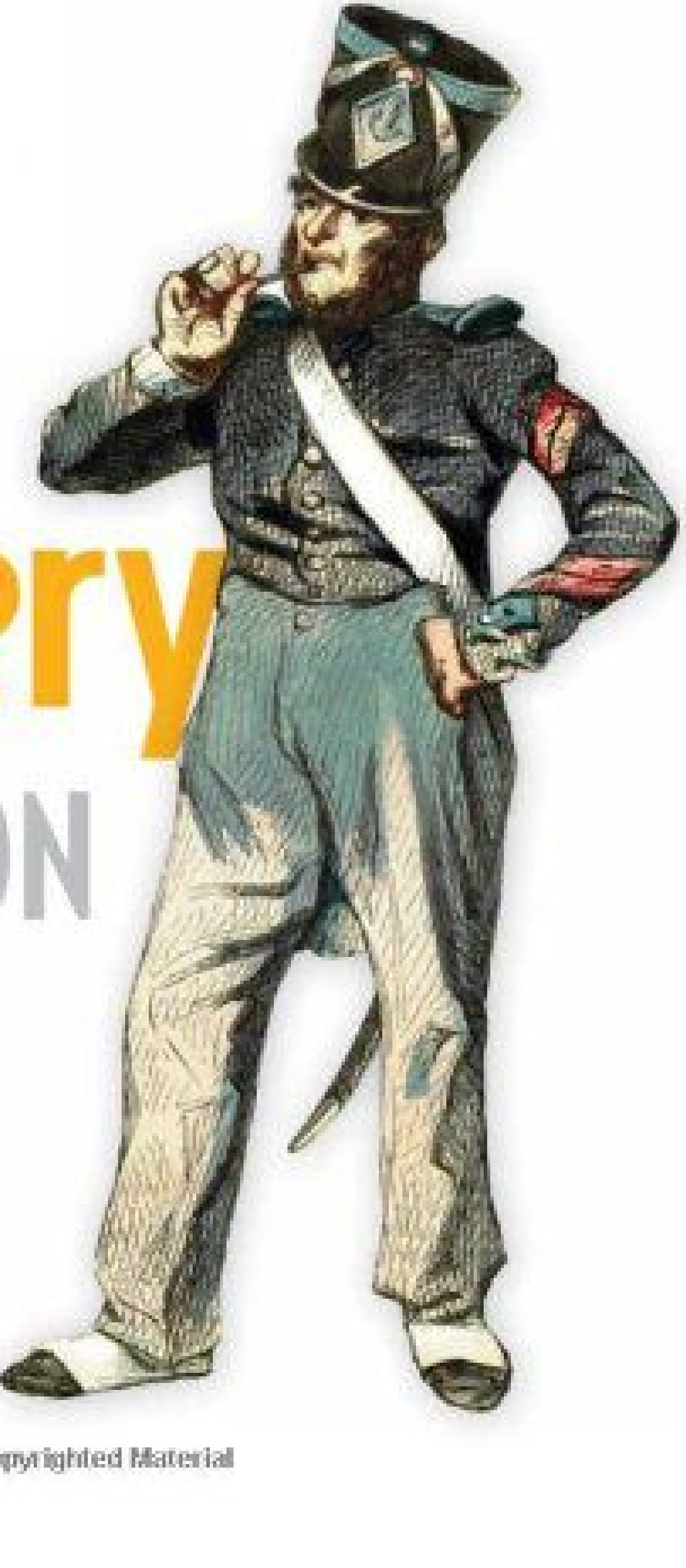
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8

jQuery IN ACTION

SECOND EDITION



MANNING

Table of Contents

4 Events are where it happens!

This chapter covers

- The event models as implemented by the browsers
- The jQuery Event Model
- Binding event handlers to DOM elements
- The `Event` object instance
- Triggering event handlers under script control
- Registering proactive event handlers

Anyone familiar with the Broadway show *Cabaret*, or its subsequent Hollywood film, probably remembers the song “Money Makes the World Go Around.” Although this cynical view might be applicable to the physical world, in the virtual realm of the World Wide Web, it’s events that make it all happen!

Like many other GUI management systems, the interfaces presented by HTML web pages are *asynchronous* and *event-driven* (even if the HTTP protocol used to deliver them to the browser is wholly synchronous in nature). Whether a GUI is implemented as a desktop program using Java Swing, X11, or the .NET Framework, or as a page in a web application using HTML and JavaScript, the program steps are pretty much the same:

1. Set up the user interface.
2. Wait for something interesting to happen.
3. React accordingly.
4. Go to 2.

The first step sets up the *display* of the user interface; the others define its *behavior*. In web pages, the browser handles the setup of the display in response to the markup (HTML and CSS) that we send to it. The script we include in the page defines the behavior of the interface.

This script takes the form of *event handlers*, also known as *listeners*, that react to the various events that occur while the page is displayed. These events could be generated by the system (timers or the completion of asynchronous requests, for example) but are most often the result of some user activity (such as moving or clicking the mouse, entering text via the keyboard, or even iPhone gestures). Without the ability to react to these events, the World Wide Web's greatest use might be limited to showing pictures of kittens.

Although HTML itself *does* define a small set of built-in semantic actions that require no scripting on our part (such as reloading pages as the result of clicking an anchor tag or submitting a form via a submit button), any other behaviors that we wish our pages to exhibit require us to handle the various events that occur as our users interact with those pages.

In this chapter, we'll examine the various ways that browsers expose these events, how they allow us to establish handlers to control what happens when these events occur, and the challenges that we face due to the multitude of differences between the browser event models. Then we'll see how jQuery cuts through the browser-induced fog to relieve us of these burdens.

Let's start off by examining how browsers expose their event models.

JavaScript you need to know!

One of the great benefits that jQuery brings to web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script ourselves. jQuery handles the nuts-and-bolts details so that we can concentrate on the job of making our applications do what they need to do!

Up to this point, the ride has been pretty painless. You only needed rudimentary JavaScript skills to code and understand the jQuery examples we introduced in the previous chapters. In this chapter and the chapters that follow, you *must* understand a handful of important fundamental JavaScript concepts to make effective use of the jQuery library.

continued

Depending on your background, you may already be familiar with these concepts, but some page authors may have been able to get pretty far without a firm grasp of these concepts—the very flexibility of JavaScript makes such a situation possible. Before we proceed, it's time to make sure that you've wrapped your head around these core concepts.

If you're already comfortable with the workings of the JavaScript `Object` and `Function` classes, and have a good handle on concepts like *function contexts* and *closures*, you may want to continue reading this and the upcoming chapters. If these concepts are unfamiliar or hazy, we strongly urge you to turn to the appendix to help you get up to speed on these necessary concepts.

4.1 Understanding the browser event models

Long before anyone considered standardizing how browsers would handle events, Netscape Communications Corporation introduced an event-handling model in its Netscape Navigator browser; all modern browsers still support this model, which is probably the best understood and most employed by the majority of page authors.

This model is known by a few names. You may have heard it termed the Netscape Event Model, the Basic Event Model, or even the rather vague Browser Event Model, but most people have come to call it the *DOM Level 0 Event Model*.

NOTE The term *DOM level* is used to indicate what level of requirements an implementation of the W3C DOM specification meets. There isn't a DOM Level 0, but that term is used to informally describe what was implemented *prior* to DOM Level 1.

The W3C didn't create a standardized model for event handling until DOM Level 2, introduced in November 2000. This model enjoys support from all modern standards-compliant browsers such as Firefox, Camino (as well as other Mozilla browsers), Safari, and Opera. Internet Explorer continues to go its own way and supports a subset of the functionality in the DOM Level 2 Event Model, albeit using a proprietary interface.

Before we see how jQuery makes that irritating fact a non-issue, let's spend some time getting to know how the various event models operate.

4.1.1 The DOM Level 0 Event Model

The DOM Level 0 Event Model is the event model that most web developers employ on their pages. In addition to being somewhat browser-independent, it's fairly easy to use.

Under this event model, event handlers are declared by assigning a reference to a function instance to properties of the DOM elements. These properties are defined to handle a specific event type; for example, a click event is handled by assigning a function to the `onclick` property, and a mouseover event by assigning a function to the `onmouseover` property of elements that support these event types.

The browsers allow us to specify the body of an event handler function as attribute values embedded within the HTML markup of the DOM elements, providing a shorthand for creating event handlers. An example of defining such handlers is shown in listing 4.1. This page can be found in the downloadable code for this book in the file chapter4/dom.0.events.html.

Listing 4.1 Declaring DOM Level 0 event handlers

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 0 Events Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.min.js"></
     script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></
     script>
    <script type="text/javascript">
      $(function(){
        $('#example')[0].onmouseover = function(event) {
          say('Crackle!');
        };
      });
    </script>
  </head>

  <body>
    <img id="example" src="example.jpg"
         onclick="say('BOOM!');"
         alt="ooooh! ahhhh!"/>
  </body>
</html>
```

❶ Defines the mouseover handler

❷ Emits text to "console"

❸ Instruments <img> element

In this example, we employ both styles of event handler declaration: declaring under script control and declaring in a markup attribute.

The page first declares a ready handler in which a reference to the image element with the id of example is obtained (using jQuery), and its onmouseover property is set to an inline function ❶. This function becomes the event handler for the element when a mouseover event is triggered on it. Note that this function expects a single parameter to be passed to it. We'll learn more about this parameter shortly.

Within this function, we employ the services of a small utility function, say() ❷, that we use to emit text messages to a dynamically created <div> element on the page that we'll call the "console." This function is declared within the imported support script file (jqia2.support.js), and will save us the trouble of using annoying and disruptive alerts to indicate when things happen on our page. We'll be using this handy function in many of the examples throughout the remainder of the book.

In the body of the page, we define an <img> element upon which we're defining the event handlers. We've already seen how to define a handler under script control in the ready handler ❶, but here we declare a handler for a click event using the onclick attribute ❸ of the <img> element.

NOTE Obviously we've thrown the concept of Unobtrusive JavaScript out the kitchen window for this example. Long before we reach the end of this chapter, we'll see why we won't need to embed event behavior in the DOM markup anymore!

Loading this page into a browser (found in the file chapter4/dom.0.events.html), waving the mouse pointer over the image a few times, and then clicking the image, results in a display similar to that shown in figure 4.1.

We declared the click event handler in the `<img>` element markup using the following attribute:

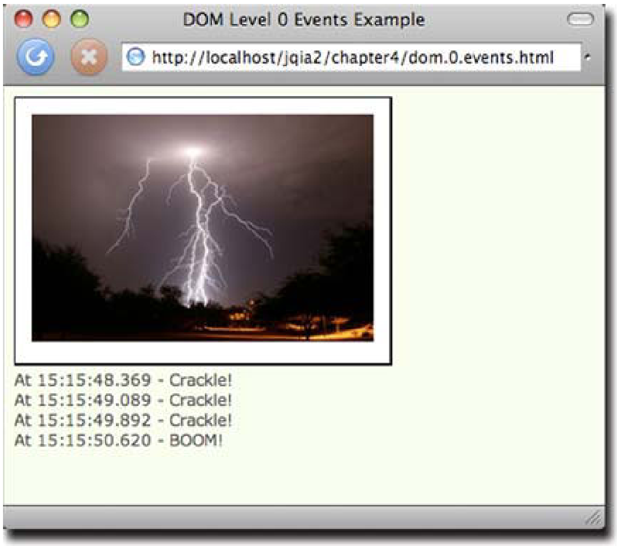
```
onclick="say('BOOM!');"
```

This might lead us to believe that the `say()` function becomes the click event handler for the element, but that's not really the case. When handlers are declared via HTML markup attributes, an anonymous function is automatically created using the value of the attribute as the function *body*. Assuming that `imageElement` is a reference to the image element, the construct created as a result of the attribute declaration is equivalent to the following:

```
imageElement.onclick = function(event) {
  say('BOOM!');
};
```

Note how the value of the attribute is used as the body of the generated function, and note that the function is created so that the `event` parameter is available within the generated function.

Before we move on to examining what that `event` parameter is all about, we should note that using the attribute mechanism of declaring DOM Level 0 event handlers violates the precepts of Unobtrusive JavaScript that we explored in chapter 1. When using jQuery in our pages, we should adhere to the principles of Unobtrusive


DOM Level 0 Events Example
http://localhost/jqia2/chapter4/dom.0.events.html

```
At 15:15:48.369 - Crackle!
At 15:15:49.089 - Crackle!
At 15:15:49.892 - Crackle!
At 15:15:50.620 - BOOM!
```

Figure 4.1 Waving the mouse over the image and clicking it results in the event handlers firing and emitting their messages to the console.

JavaScript and avoid mixing behavior defined by JavaScript with display markup. We'll shortly see that jQuery provides a much better way to declare event handlers than either of these means.

But first, let's examine what that `event` parameter is all about.

THE EVENT INSTANCE

When an event handler is fired, an instance of a class named `Event` is passed to the handler as its first parameter in most browsers. Internet Explorer, always the life of the party, does things in its own proprietary way by tacking the `Event` instance onto a global property (in other words, a property on `window`) named `event`.

In order to deal with this discrepancy, we'll often see the following used as the first statement in a non-jQuery event handler:

```
if (!event) event = window.event;
```

This levels the playing field by using feature detection (a concept we'll explore in greater depth in chapter 6) to check if the `event` parameter is `undefined` (or `null`) and assigning the value of the window's `event` property to it if so. After this statement, the `event` parameter can be referenced regardless of how it was made available to the handler.

The properties of the `Event` instance provide a great deal of information regarding the event that has been fired and is currently being handled. This includes details such as which element the event was triggered on, the coordinates of mouse events, and which key was clicked for keyboard events.

But not so fast. Not only does Internet Explorer use a proprietary means to get the `Event` instance to the handler, but it also uses a proprietary definition of the `Event` class in place of the W3C-defined standard—we're not out of the object-detection woods yet.

For example, to get a reference to the target element—the element on which the event was triggered—we access the `target` property in standards-compliant browsers and the `srcElement` property in Internet Explorer. We deal with this inconsistency by employing feature detection with a statement such as the following:

```
var target = (event.target) ? event.target : event.srcElement;
```

This statement tests whether `event.target` is defined and, if so, assigns its value to the local `target` variable; otherwise, it assigns `event.srcElement`. We'll be required to take similar steps for other `Event` properties of interest.

Up until this point, we've acted as if event handlers are only pertinent to the elements that serve as the trigger to an event—the image element of listing 4.1, for example—but events propagate throughout the DOM tree. Let's find out about that.

EVENT BUBBLING

When an event is triggered on an element in the DOM tree, the event-handling mechanism of the browser checks to see if a handler has been established for that particular event on that element and, if so, invokes it. But that's hardly the end of the story.

After the target element has had its chance to handle the event, the event model checks with the parent of that element to see if *it* has established a handler for the event type, and if so, it's also invoked—after which *its* parent is checked, then its parent, then its parent, and on and on, all the way up to the top of the DOM tree. Because the event handling propagates upward like the bubbles in a champagne flute (assuming we view the DOM tree with its root at the top), this process is termed *event bubbling*.

Let's modify the example from listing 4.1 so that we can see this process in action. Consider the code in listing 4.2.

Listing 4.2 Events propagate from the point of origin to the top of the DOM

```
<!DOCTYPE html>
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 0 Bubbling Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></
     script>
    <script type="text/javascript">
      $(function(){                                      ❶ Selects every
        $('*').each(function(){                            element on the page
          var current = this;
          this.onclick = function(event) {                   Applies onclick handler to
            if (!event) event = window.event;              ❷ every selected element
            var target = (event.target) ?
                         event.target : event.srcElement;
            say('For ' + current.tagName + '#'+ current.id +
                ' target is ' +
                target.tagName + '#' + target.id);
          };
        });
      });
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="example" src="example.jpg" alt="ooooh! ahhhh!"/>
      </div>
    </div>
  </body>
</html>
```

We do a lot of interesting things in the changes to this example. First, we remove the previous handling of the mouseover event so that we can concentrate on the click event. We also embed the image element that will serve as the target for our event experiment in a couple of nested `<div>` elements, merely to place the image element artificially deeper within the DOM hierarchy. We also give almost every element in the page a specific and unique `id`—even the `<body>` and `<html>` tags!

Now let's look at even more interesting changes.

In the ready handler for the page, we use jQuery to select all elements on the page and to iterate over each one with the `each()` method ❶. For each matched element, we record its instance in the local variable `current` and establish an `onclick` handler ❷. This handler first employs the browser-dependent tricks that we discussed in the previous section to locate the `Event` instance and identify the event target, and then emits a console message. This message is the most interesting part of this example.

It displays the tag name and `id` of the current element, putting *closures* to work (please read the section on closures in the appendix if closures are a subject that gives you heartburn), followed by the `id` of the target. By doing so, each message that's logged to the console displays the information about the current element of the bubble process, as well as the target element that started the whole shebang.

Loading the page (located in the file chapter4/dom.0.propagation.html) and clicking the image results in the display of figure 4.2.

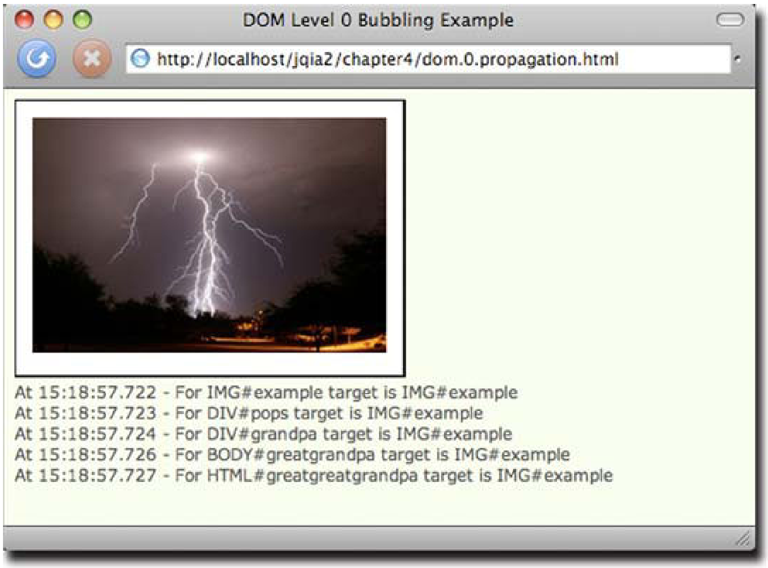
This clearly illustrates that, when the event is fired, it's delivered first to the target element and then to each of its ancestors in turn, all the way up to the root `<html>` element.

This is a powerful ability because it allows us to establish handlers on elements at any level to handle events occurring on its descendents. Consider a handler on a `<form>` element that reacts to any change event on its child elements to effect dynamic changes to the display based upon the elements' new values.

But what if we don't *want* the event to propagate? Can we stop it?

AFFECTING EVENT PROPAGATION AND SEMANTIC ACTIONS

There may be occasions when we want to prevent an event from bubbling any further up the DOM tree. This might be because we're fastidious and we know that we've already accomplished any processing necessary to handle the event, or we may want to forestall unwanted handling that might occur higher up in the chain.


DOM Level 0 Bubbling Example
http://localhost/jqia2/chapter4/dom.0.propagation.html

```
At 15:18:57.722 - For IMG#example target is IMG#example
At 15:18:57.723 - For DIV#pops target is IMG#example
At 15:18:57.724 - For DIV#grandpa target is IMG#example
At 15:18:57.726 - For BODY#greatgrandpa target is IMG#example
At 15:18:57.727 - For HTML#greatgreatgrandpa target is IMG#example
```

Figure 4.2 The console messages clearly show the propagation of the event as it bubbles up the DOM tree from the target element to the tree root.

Regardless of the reason, we can prevent an event from propagating any higher via mechanisms provided on the `Event` instance. For standards-compliant browsers, we call the `stopPropagation()` method of the `Event` instance to halt the propagation of the event further up the ancestor hierarchy. In Internet Explorer, we set a property named `cancelBubble` to `true` in the `Event` instance. Interestingly, many modern standards-compliant browsers support the `cancelBubble` mechanism even though it's not part of any W3C standard.

Some events have default semantics associated with them. As examples, a click event on an anchor element will cause the browser to navigate to the element's `href`, and a submit event on a `<form>` element will cause the form to be submitted. Should we wish to cancel these semantic actions—sometimes termed the *default actions*—of the event, we simply return the value `false` from the event handler.

A frequent use for such an action is in the realm of form validation. In the handler for the form's submit event, we can make validation checks on the form's controls and return `false` if any problems with the data entry are detected.

You may also have seen the following on `<form>` elements:

```
<form name="myForm" onsubmit="return false;" ...
```

This effectively prevents the form from being submitted in any circumstances except under script control (via `form.submit()`, which doesn't trigger a submit event)—a common trick used in many Ajax applications where asynchronous requests will be made in lieu of form submissions.

Under the DOM Level 0 Event Model, almost every step we take in an event handler involves using browser-specific detection in order to figure out what action to take. What a headache! But don't put away the aspirin yet—it doesn't get any easier when we consider the more advanced event model.

4.1.2 The DOM Level 2 Event Model

One severe shortcoming of the DOM Level 0 Event Model is that, because a property is used to store a reference to a function that's to serve as an event handler, only one event handler per element can be registered for any specific event type at a time. If we have two things that we want to do when an element is clicked, the following statements aren't going to let that happen:

```
someElement.onclick = doFirstThing;
someElement.onclick = doSecondThing;
```

Because the second assignment replaces the previous value of the `onclick` property, only `doSecondThing` is invoked when the event is triggered. Sure, we could wrap both functions in another single function that calls both, but as pages get more complicated, as is quite likely in highly interactive applications, it becomes increasingly difficult to keep track of such things. Moreover, if we use multiple reusable components or libraries in a page, they may have no idea of the event-handling needs of the other components.

We could employ other solutions: implementing the Observable pattern, which establishes a publish/subscribe scheme for the handlers, or even tricks using closures. But all of these add complexity to pages that are likely to already be complex enough.

Besides the establishment of a *standard* event model, the DOM Level 2 Event Model was designed to address these types of problems. Let's see how event handlers, even multiple handlers, are established on DOM elements under this more advanced model.

ESTABLISHING EVENT HANDLERS

Rather than assigning a function reference to an element property, DOM Level 2 event handlers—also termed *listeners*—are established via an element *method*. Each DOM element defines a method named `addEventListener()` that's used to attach event handlers (listeners) to the element. The format of this method is as follows:

```
addEventListener(eventType,listener,useCapture)
```

The `eventType` parameter is a string that identifies the type of event to be handled. These string values are, generally, the same event names we used in the DOM Level 0 Event Model without the *on* prefix: for example: click, mouseover, keydown, and so on.

The `listener` parameter is a reference to the function (or an inline function) that's to be established as the handler for the named event type on the element. As in the basic event model, the `Event` instance is passed to this function as its first parameter.

The final parameter, `useCapture`, is a Boolean whose operation we'll explore in a few moments, when we discuss event propagation in the Level 2 Model. For now, we'll leave it set to `false`.

Let's once again change the example from listing 4.1 to use the more advanced event model. We'll concentrate only on the click event type; this time, we'll establish *three* click event handlers on the image element. The new example code can be found in the file chapter4/dom.2.events.html and is shown in listing 4.3.

Listing 4.3 Establishing event handlers with the DOM Level 2 Event Model

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 2 Events Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></
     script>
    <script type="text/javascript">                      1 Establishes three
      $(function(){                                         event handlers!
        var element = $('#example')[0];
        element.addEventListener('click',function(event) {
          say('BOOM once!');
        },false);
        element.addEventListener('click',function(event) {
          say('BOOM twice!');
        },false);
    element.addEventListener('click',function(event) {
     say('BOOM three times!');
    },false);
   });
  </script>
 </head>

 <body>
  <img id="example" src="example.jpg"/>
 </body>
</html>
```

This code is simple, but it clearly shows how we can establish multiple event handlers on the same element for the same event type—something we were not able to do easily with the Basic Event Model. In the ready handler for the page ❶, we grab a reference to the image element and then establish *three* event handlers for the click event.

Loading this page into a standards-compliant browser (*not* Internet Explorer) and clicking the image results in the display shown in figure 4.3.

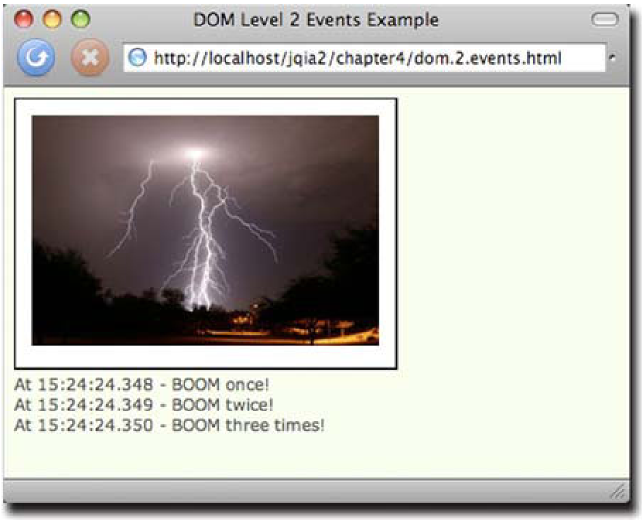
Note that even though the handlers fire in the order in which they were established, *this order isn't guaranteed by the standard*! Testers of this code never observed an order other than the order of establishment, but it would be foolish to write code that relies on this order. Always be aware that multiple handlers established on an element may fire in random order.

Now, let's find out what's up with that `useCapture` parameter.

EVENT PROPAGATION

We saw earlier that, with the Basic Event Model, once an event was triggered on an element the event propagated from the target element upwards in the DOM tree to all the target's ancestors. The advanced Level 2 Event Model also provides this bubbling phase but ups the ante with an additional capture phase.

Under the DOM Level 2 Event Model, when an event is triggered, the event first propagates from the root of the DOM tree down to the target element and then


DOM Level 2 Events Example
http://localhost/jqia2/chapter4/dom.2.events.html

```
At 15:24:24.348 - BOOM once!
At 15:24:24.349 - BOOM twice!
At 15:24:24.350 - BOOM three times!
```

Figure 4.3 Clicking the image once demonstrates that all three handlers established for the click event are triggered.

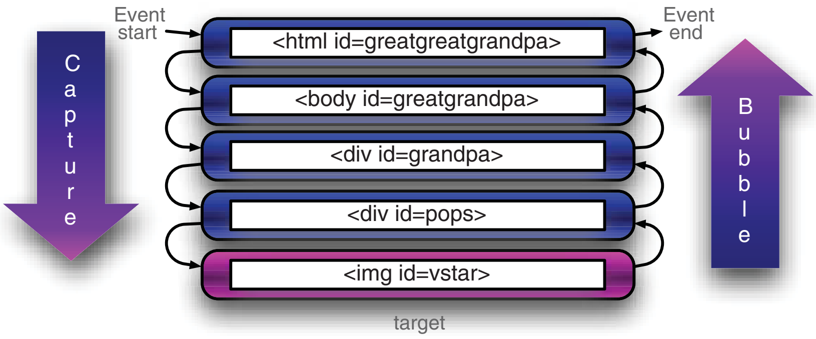


Figure 4.4 Propagation in the DOM Level 2 Event Model traverses the DOM hierarchy twice: once from top to target during capture phase and once from target to top during bubble phase.

propagates again from the target element up to the DOM root. The former phase (root to target) is called *capture phase*, and the latter (target to root) is called *bubble phase*.

When a function is established as an event handler, it can be flagged as a capture handler, in which case it will be triggered during capture phase, or as a bubble handler, to be triggered during bubble phase. As you might have guessed by this time, the `useCapture` parameter to `addEventListener()` identifies which type of handler is established. A value of `false` for this parameter establishes a bubble handler, whereas a value of `true` registers a capture handler.

Think back a moment to the example of listing 4.2 where we explored the propagation of the Basic Model events through a DOM hierarchy. In that example, we embedded an image element within two layers of `<div>` elements. Within such a hierarchy, the propagation of a click event with the `<img>` element as its target would move through the DOM tree as shown in figure 4.4.

Let's put that to the test, shall we? Listing 4.4 shows the code for a page containing the element hierarchy of figure 4.4 (chapter4/dom.2.propagation.html).

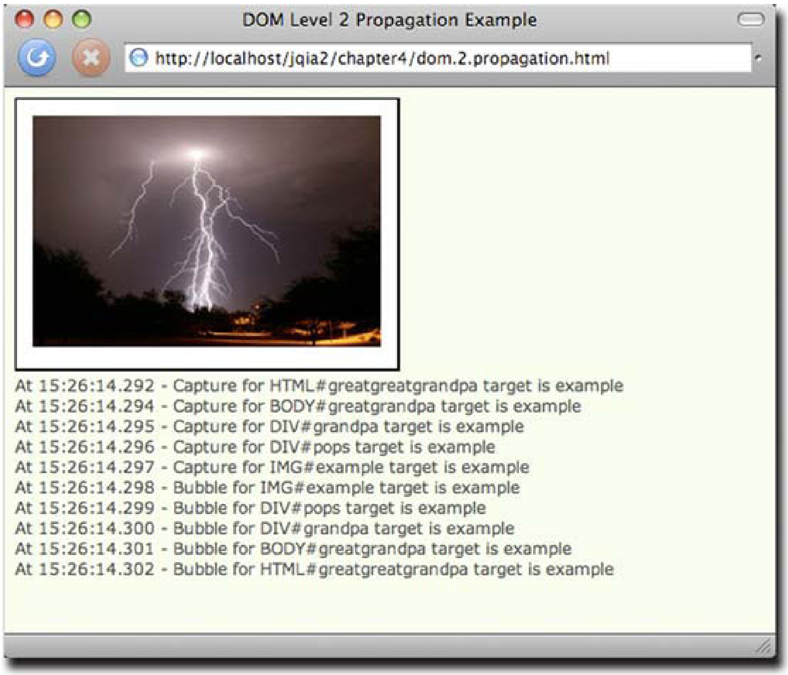
Listing 4.4 Tracking event propagation with bubble and capture handlers

```
<!DOCTYPE html>
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 2 Propagation Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></
     script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){                        ❶ Establishes listeners
          var current = this;                            on all elements
          this.addEventListener('click',function(event) {
            say('Capture for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
          },true);
          this.addEventListener('click',function(event) {
            say('Bubble for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
            },false);
        });
      });
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="example" src="example.jpg"/>
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>
```

This code changes the example of listing 4.2 to use the DOM Level 2 Event Model API to establish the event handlers. In the ready handler ❶, we use jQuery's powerful abilities to run through every element of the DOM tree. On each, we establish two handlers: one capture handler and one bubble handler. Each handler emits a message to the console identifying which type of handler it is, the current element, and the `id` of the target element.

With the page loaded into a standards-compliant browser, clicking the image results in the display in figure 4.5, showing the progression of the event through the handling phases and the DOM tree. Note that, because we defined both capture and bubble handlers for the target, two handlers were executed for the target and all its ancestor nodes.


DOM Level 2 Propagation Example
http://localhost/jqia2/chapter4/dom.2.propagation.html

```
At 15:26:14.292 - Capture for HTML#greatgreatgrandpa target is example
At 15:26:14.294 - Capture for BODY#greatgrandpa target is example
At 15:26:14.295 - Capture for DIV#grandpa target is example
At 15:26:14.296 - Capture for DIV#pops target is example
At 15:26:14.297 - Capture for IMG#example target is example
At 15:26:14.298 - Bubble for IMG#example target is example
At 15:26:14.299 - Bubble for DIV#pops target is example
At 15:26:14.300 - Bubble for DIV#grandpa target is example
At 15:26:14.301 - Bubble for BODY#greatgrandpa target is example
At 15:26:14.302 - Bubble for HTML#greatgreatgrandpa target is example
```

Figure 4.5 Clicking the image results in each handler emitting a console message that identifies the path of the event during both capture and bubble phases.

Well, now that we've gone through all the trouble to understand these two types of handlers, we should know that capture handlers are hardly ever used in web pages. The simple reason for that is that Internet Explorer doesn't support the DOM Level 2 Event Model. Although it *does* have a proprietary model corresponding to the bubble phase of the Level 2 standard, it doesn't support any semblance of a capture phase.

Before we look at how jQuery can help sort all this mess out, let's briefly examine the Internet Explorer Model.

4.1.3 The Internet Explorer Event Model

Internet Explorer (IE 6, IE 7, and, most disappointingly, even IE 8) doesn't provide support for the DOM Level 2 Event Model. These versions of Microsoft's browser provide a proprietary interface that closely resembles the bubble phase of the standard model.

Rather than `addEventListener()`, the Internet Explorer Model defines a method named `attachEvent()` for each DOM element. This method accepts two parameters similar to those of the standard model:

```
attachEvent(eventName,handler)
```

The first parameter is a string that names the event type to be attached. The standard event names aren't used; the name of the corresponding element property from the DOM Level 0 Model is used—"onclick", "onmouseover", "onkeydown", and so on.

The second parameter is the function to be established as the handler, and as in the Basic Model, the `Event` instance must be fetched from the `window.event` property.

What a mess! Even when using the relatively browser-independent DOM Level 0 Model, we're faced with a tangle of browser-dependent choices to make at each stage of event handling. And when using the more capable DOM Level 2 or Internet Explorer Model, we even have to diverge our code when establishing the handlers in the first place.

Well, jQuery is going to make our lives simpler by hiding these browser disparities from us as much as it possibly can. Let's see how!

4.2 The jQuery Event Model

Although it's true that the creation of highly interactive applications requires a hefty reliance on event handling, the thought of writing event-handling code on a large scale while dealing with the browser differences is enough to daunt even the most intrepid of page authors.

We could hide the differences behind an API that abstracts the differences away from our page code, but why bother when jQuery has already done it for us?

jQuery's event model implementation, which we'll refer to informally as the jQuery Event Model, exhibits the following features:

- Provides a unified method for establishing event handlers
- Allows multiple handlers for each event type on each element

- Uses standard event-type names: for example, click or mouseover
- Makes the `Event` instance available as a parameter to the handlers
- Normalizes the `Event` instance for the most-often-used properties
- Provides unified methods for event canceling and default action blocking

With the notable exception of support for a capture phase, the feature set of the jQuery Event Model closely resembles that of the DOM Level 2 Event Model while supporting both standards-compliant browsers and Internet Explorer with a single API. The omission of the capture phase should not be an issue for the vast majority of page authors who never use it (or even know it exists) due to its lack of support in IE.

Is it really that simple? Let's find out.

4.2.1 Binding event handlers with jQuery

Using the jQuery Event Model, we can establish event handlers on DOM elements with the `bind()` method. Consider the following simple example:

```
$('img').bind('click',function(event){alert('Hi there!');});
```

This statement binds the supplied inline function as the click event handler for every image on a page. The full syntax of the `bind()` method is as follows:

Method syntax: bind

`bind(eventType,data,handler)`
`bind(eventMap)`

Establishes a function as the event handler for the specified event type on all elements in the matched set.

Parameters

`eventType` (String) Specifies the name of the event type or types for which the handler is to be established. Multiple event types can be specified as a space-separated list.
These event types can be namespaced with a suffix affixed to the event name with a period character. See the remainder of this section for details.

`data` (Object) Caller-supplied data that's attached to the `Event` instance as a property named `data` and made available to the handler function. If omitted, the handler function can be specified as the second parameter.

`handler` (Function) The function that's to be established as the event handler. When invoked, it will be passed the `Event` instance, and its function context (`this`) is set to the current element of the bubble phase.

`eventMap` (Object) A JavaScript object that allows handlers for multiple event types to be established in a single call. The property names identify the event type (same as would be used for the `eventType` parameter), and the property value provides the handler.

Returns

The wrapped set.

Let's put `bind()` into action. Taking the example of listing 4.3 and converting it from the DOM Level 2 Event Model to the jQuery Event Model, we end up with the code shown in listing 4.5 and found in the file chapter4/jquery.events.html.

Listing 4.5 Establishing advanced event handlers without browser-specific code

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery Events Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></
     script>
    <script type="text/javascript">
      $(function(){
        $('#example')
          .bind('click',function(event) {
            say('BOOM once!');
          })
          .bind('click',function(event) {
            say('BOOM twice!');
          })
          .bind('click',function(event) {
            say('BOOM three times!');
          });
      });
    </script>
  </head>

  <body>
    <img id="example" src="example.jpg"/>
  </body>
</html>
```

❶ Binds three event handlers to the image

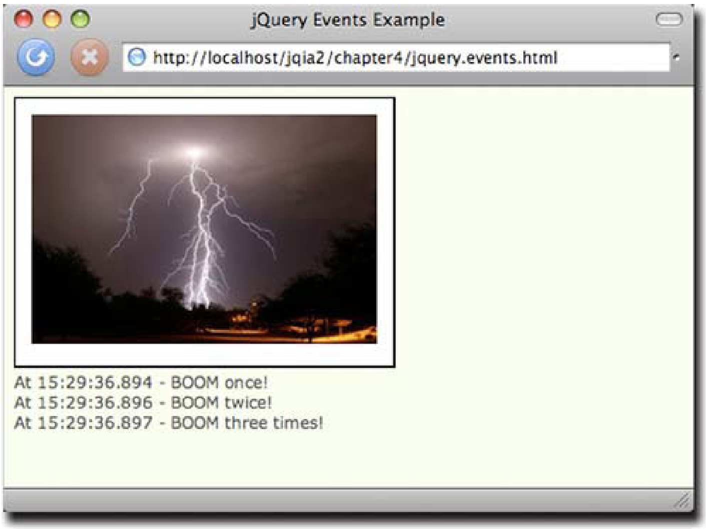
The changes to this code, limited to the body of the ready handler, are minor but significant ❶. We create a wrapped set consisting of the target `<img>` element and apply three `bind()` methods to it—remember, jQuery chaining lets us apply multiple methods in a single statement—each of which establishes a click event handler on the element.

Loading this page into a standards-compliant browser and clicking the image results in the display of figure 4.6, which, not surprisingly, is the exact same result we saw in figure 4.3 (except for the URL and window caption).

But perhaps more importantly, it also works when loaded into Internet Explorer, as shown in figure 4.7. This was not possible using the code from listing 4.3 without adding browser-specific testing and branching code to use the correct event model for the current browser.

At this point, page authors who have wrestled with mountains of browser-specific event-handling code in their pages are no doubt singing “Happy Days Are Here Again” and spinning in their office chairs. Who could blame them?

Another nifty little event-handling extra that jQuery provides for us is the ability to group event handlers by assigning them to a namespace. Unlike conventional namespacing (which assigns namespaces via a prefix), the event names are namespaced by adding a *suffix* to the event name separated by a period character. In



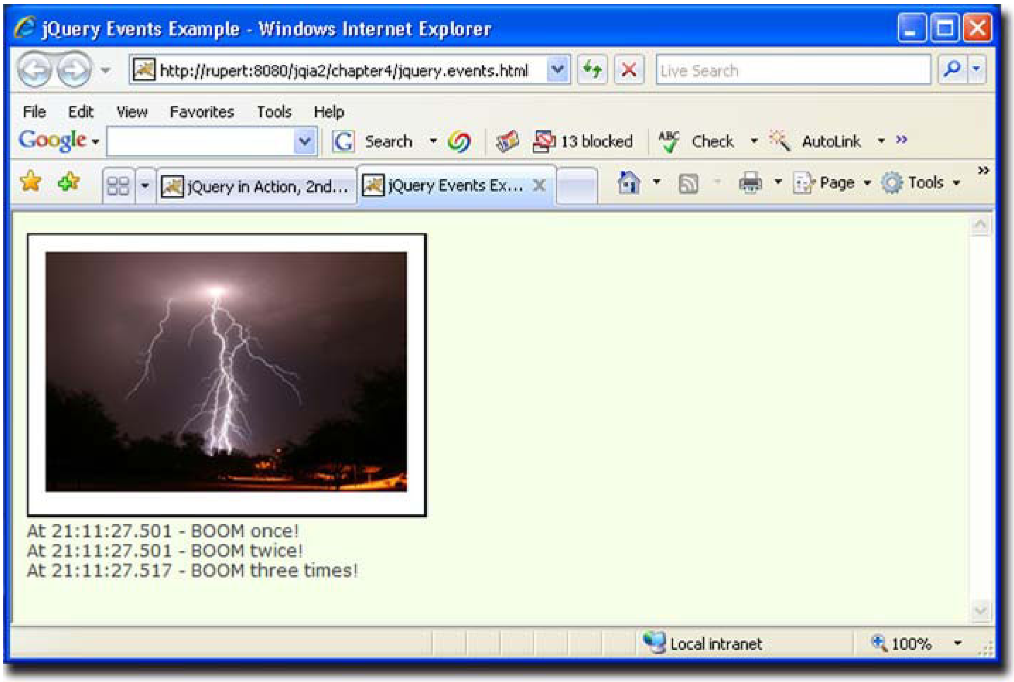
```
At 15:29:36.894 - BOOM once!
At 15:29:36.896 - BOOM twice!
At 15:29:36.897 - BOOM three times!
```

Figure 4.6 Using the jQuery Event Model allows us to specify multiple event handlers just like the DOM Level 2 Event Model.

fact, if you'd like, you can use multiple suffixes to place the event into multiple namespaces.

By grouping event bindings in this way, we can easily act upon them later as a unit.

Take, for example, a page that has two modes: a display mode and an edit mode. When in edit mode, event listeners are placed on many of the page elements, but these listeners aren't appropriate for display mode and need to be removed when the


jQuery Events Example - Windows Internet Explorer
http://rupert:8080/jqia2/chapter4/jquery.events.html
Live Search
File Edit View Favorites Tools Help
Google
Search
13 blocked
Check
AutoLink
jQuery in Action, 2nd...
jQuery Events Ex...
Page
Tools
Local intranet
100%

```
At 21:11:27.501 - BOOM once!
At 21:11:27.501 - BOOM twice!
At 21:11:27.517 - BOOM three times!
```

Figure 4.7 The jQuery Event Model allows us to use a unified events API to support events across the standards-compliant browsers as well as Internet Explorer.

page transitions out of edit mode. We could namespace the edit mode events with code such as this:

```
$('#thing1').bind('click.editMode',someListener);
$('#thing2').bind('click.editMode',someOtherListener);
  ...
$('#thingN').bind('click.editMode',stillAnotherListener);
```

By grouping all these bindings into a namespace named `editMode`, we can later operate upon them as a whole. For example, when the page leaves edit mode and it comes time to remove all the bindings, we could do this easily with

```
$('*').unbind('click.editMode');
```

This will remove all `click` bindings (the explanation of the `unbind()` method is coming up in the next section) in the namespace `editMode` for all elements on the page.

Before we leave `bind()`, let's consider one more example:

```
$('.whatever').bind({
  click: function(event) { /* handle clicks */ },
  mouseenter: function(event) { /* handle mouseenters */ },
  mouseleave: function(event) { /* handle mouseleaves */ }
});
```

For occasions when we want to bind multiple event types to an element, we can do it with a single call to `bind()`, as shown.

In addition to the `bind()` method, jQuery provides a handful of shortcut methods to establish specific event handlers. Because the syntax of each of these methods is identical except for the name of the method, we'll save some space and present them all in the following single syntax descriptor:

Method syntax: specific event binding

***eventTypeName*(listener)**

Establishes the specified function as the event handler for the event type named by the method's name. The supported methods are as follows:

```
blur
change
click
dblclick
error
focus
focusin
focusout
keydown
keypress
keyup
load
mousedown
mouseenter
mouseleave
mousemove
mouseout
mouseover
mouseup
ready
resize
scroll
select
submit
unload
```

Note that when using these shortcut methods, we can't specify a data value to be placed in the `event.data` property.

Parameters

`listener`	(Function) The function that's to be established as the event handler.

Returns

The wrapped set.

The focusin and focusout events deserve some discussion.

It's not hard to imagine scenarios where we'd want to handle focus and blur events in a central manner. For example, let's say that we wanted to keep track of which fields in a form had been visited. Rather than establishing a handler on each and every element, it'd be handy to establish a single handler on the form. But we can't.

The focus and blur events, by their nature, do not bubble up the DOM tree. Therefore, a focus handler established on the form element would never get invoked.

This is where the focusin and focusout events come in. Handlers established for these events on focusable elements are invoked whenever the element receives or loses focus, as are any such handlers established on the ancestors of the focusable element.

jQuery also provides a specialized version of the `bind()` method, named `one()`, that establishes an event handler as a one-shot deal. Once the event handler executes the first time, it's automatically removed as an event handler. Its syntax is similar to the `bind()` method:

Method syntax: one

`one(eventType,data,listener)`

Establishes a function as the event handler for the specified event type on all elements in the matched set. Once executed, the handler is automatically removed.

Parameters

`eventType`	(String) Specifies the name of the event type for which the handler is to be established.
`data`	(Object) Caller-supplied data that's attached to the `Event` instance for availability to the handler function. If omitted, the handler function can be specified as the second parameter.
`listener`	(Function) The function that's to be established as the one-time event handler.

Returns

The wrapped set.

These methods give us many choices for binding an event handler to matched elements. And once a handler is bound, we may eventually need to remove it.

4.2.2 *Removing event handlers*

Typically, once an event handler is established, it remains in effect for the remainder of the life of the page. But particular interactions may dictate that handlers be removed based on certain criteria. Consider, for example, a page where multiple steps are presented, and once a step has been completed, its controls revert to read-only.

For such cases, it would be advantageous to remove event handlers under script control. We've seen that the `one()` method can automatically remove a handler after it has completed its first (and only) execution, but for the more general case where we'd like to remove event handlers under our own control, jQuery provides the `unbind()` method.

The syntax of `unbind()` is as follows:

Method syntax: unbind

`unbind(eventType,listener)`
`unbind(event)`

Removes events handlers from all elements of the wrapped set as specified by the optional passed parameters. If no parameters are provided, all listeners are removed from the elements.

Parameters

`eventType` (String) If provided, specifies that only listeners established for the specified event type are to be removed.

`listener` (Function) If provided, identifies the specific listener that's to be removed.

`event` (Event) Removes the listener that triggered the event described by this `Event` instance.

Returns

The wrapped set.

This method can be used to remove event handlers from the elements of the matched set at various levels of granularity. All listeners can be removed by omitting any parameters, or listeners of a specific type can be removed by providing just that event type.

Specific handlers can be removed by providing a reference to the function originally established as the listener. To do this, a reference to the function must be retained when binding the function as an event listener in the first place. For this reason, when a function that's eventually to be removed as a handler is originally established as a listener, it's either defined as a top-level function (so that it can be referred to by its top-level variable name) or a reference to it is retained by some other means. Supplying the function as an anonymous inline reference would make it impossible to later reference the function in a call to `unbind()`.

That's a situation where using name-spaced events can come in quite handy, as you can unbind all events in a particular namespace without having to retain individual references to the listeners. For example:

```
$('*').unbind('.fred');
```

This statement will remove all event listeners in namespace `fred`.

So far, we've seen that the jQuery Event Model makes it easy to establish (as well as remove) event handlers without worries about browser differences, but what about writing the event handlers themselves?

4.2.3 Inspecting the Event instance

When an event handler established with the `bind()` method (or any of its related convenience methods) is invoked, an `Event` instance is passed to it as the first parameter to the function regardless of the browser, eliminating the need to worry about the `window.event` property under Internet Explorer. But that still leaves us dealing with the divergent properties of the `Event` instance, doesn't it?

Thankfully, no, because truth be told, jQuery doesn't really pass the `Event` instance to the handlers.

Screech! (sound of needle being dragged across record).

Yes, we've been glossing over this little detail because, up until now, it hasn't mattered. But now that we've advanced to the point where we're going to examine the instance within handlers, the truth must be told!

In reality, jQuery defines an object of type `jQuery.Event` that it passes to the handlers. But we can be forgiven our simplification, because jQuery copies most of the original `Event` properties to this object. As such, if you only look for the properties that you expected to find on Event, the object is almost indistinguishable from the original `Event` instance.

But that's not the important aspect of this object—what's really valuable, and the reason that this object exists, is that it holds a set of normalized values and methods that we can use independently of the containing browser, ignoring the differences in the `Event` instance.

Table 4.1 lists the `jQuery.Event` properties and methods that are safe to access in a platform-independent manner.

Table 4.1 Browser-independent `jQuery.Event` properties

Name	Description
PROPERTIES	
`altKey`	Set to `true` if the Alt key was pressed when the event was triggered, `false` if not. The Alt key is labeled Option on most Mac keyboards.
`ctrlKey`	Set to `true` if the Ctrl key was pressed when the event was triggered, `false` if not.
`currentTarget`	The current element during the bubble phase. This is the same object that's set as the function context of the event handler.
`data`	The value, if any, passed as the second parameter to the `bind()` method when the handler was established.
`metaKey`	Set to `true` if the Meta key was pressed when the event was triggered, `false` if not. The Meta key is the Ctrl key on PCs and the Command key on Macs.
`pageX`	For mouse events, specifies the horizontal coordinate of the event relative to the page origin.
`pageY`	For mouse events, specifies the vertical coordinate of the event relative to the page origin.
`relatedTarget`	For mouse movement events, identifies the element that the cursor left or entered when the event was triggered.
`screenX`	For mouse events, specifies the horizontal coordinate of the event relative to the screen origin.
`screenY`	For mouse events, specifies the vertical coordinate of the event relative to the screen origin.

Table 4.1 Browser-independent `jQuery.Event` properties *(continued)*

Name	Description
`shiftKey`	Set to `true` if the Shift key was pressed when the event was triggered, `false` if not.
`result`	The most recent non-undefined value returned from a previous event handler.
`target`	Identifies the element for which the event was triggered.
`timestamp`	The timestamp, in milliseconds, when the `jQuery.Event` instance was created.
`type`	For all events, specifies the type of event that was triggered (for example, "click"). This can be useful if you're using one event handler function for multiple events.
`which`	For keyboard events, specifies the numeric code for the key that caused the event; for mouse events, specifies which button was pressed (1 for left, 2 for middle, 3 for right). This should be used instead of `button`, which can't be relied on to function consistently across browsers.
METHODS	
`preventDefault()`	Prevents any default semantic action (such as form submission, link redirection, checkbox state change, and so on) from occurring.
`stopPropagation()`	Stops any further propagation of the event up the DOM tree. Additional events on the current target aren't affected. Works with browser-defined events as well as custom events.
`stopImmediatePropagation()`	Stops all further event propagation including additional events on the current target.
`isDefaultPrevented()`	Returns `true` if the `preventDefault()` method has been called on this instance.
`isPropagationStopped()`	Returns `true` if the `stopPropagation()` method has been called on this instance.
`isImmediatePropagationStopped()`	Returns `true` if the `stopImmediatePropagation()` method has been called on this instance.

It's important to note that the `keycode` property isn't reliable across browsers for nonalphabetic characters. For instance, the left arrow key has a code of 37, which works reliably on keyup and keydown events, but Safari returns nonstandard results for these keys on a keypress event.

We can get a reliable, case-sensitive character code in the `which` property of keypress events. During keyup and keydown events, we can only get a case-insensitive key code (so *a* and *A* both return 65), but we can determine case by checking `shiftKey`.

Also, if we want to stop the propagation of the event (but not *immediate* propagation), as well as cancel its default behavior, we can simply return `false` as the return value of the listener function.

All of this gives us the ability to exert fine-grained control over the establishment and removal of event handlers for all the elements that exist within the DOM; but what about elements that don't exist yet, but *will?*

4.2.4 Proactively managing event handlers

With the `bind()` and `unbind()` methods (and the plethora of convenience methods), we can readily control which event handlers are to be established on the elements of the DOM. The ready handler gives us a convenient place to initially establish handlers on the DOM elements that are created from the HTML markup on the page, or created within the ready handler.

But one of the whole reasons for using jQuery, as we saw in the last chapter, is the ease with which it allows us to dynamically manipulate the DOM. And when we throw Ajax into the mix, a subject that we'll address in chapter 8, it's likely that DOM elements will be coming into and out of existence frequently during the lifetime of the page. The ready handler isn't going to be of much help in managing the event handlers for these dynamic elements that don't exist when the ready handler is executed.

We can certainly manage event handlers on the fly as we use jQuery to manipulate the DOM, but wouldn't it be nice if we could keep all the event management code together in one place?

SETTING UP "LIVE" EVENT HANDLING

jQuery grants our wish with the `live()` method, which allows us to seemingly proactively establish event handlers for elements that don't even exist yet!

The syntax of `live()` is as follows:

Method syntax: live

`live(eventType,data,listener)`

Causes the passed listener to be invoked as a handler whenever an event identified by the event type occurs on any element that matches the selector used to create the wrapped set, regardless of whether those elements exist or not when this method is called.

Parameters

`eventType` (String) Specifies the name of the event type for which the handler is to be invoked. Unlike `bind()`, a space-separated list of event types can't be specified.

`data` (Object) Caller-supplied data that's attached to the `Event` instance as a property named `data` for availability to the handler function. If omitted, the handler function can be specified as the second parameter.

`listener` (Function) The function that's to be invoked as the event handler. When invoked, it will be passed the `Event` instance, and its function context (`this`) is set to the target element.

Returns

The wrapped set.

If the syntax of this method reminds you of the syntax for the bind() method, you'd be right. This methods looks and acts a lot like bind(), except that when a corresponding event occurs, it will be triggered for all elements that match the selector, even if those elements aren't in existence at the time that live() is called.

For example, we might write

```
$('div.attendToMe').live(
  'click',
   function(event){ alert(this); }
);
```

Throughout the lifetime of the page, a click on any <div> element with class attendToMe will result in the handler being invoked as an event handler, complete with a passed event instance. And the preceding code doesn't need to be in a ready handler because, for "live" events, it doesn't matter whether the DOM has been built yet or not.

The live() method makes it amazingly easy to establish all the event handlers needed on the page in one place and at the outset, without having to worry about whether the elements already exist or when the elements will be created.

But some cautions must be exercised when using live(). Because of its similarity to bind(), you might expect "live events" to work in exactly the same manner as native events. But there are differences that may or may not be important on your page.

Firstly, recognize that live events *aren't* native "normal" events. When an event such as a click occurs, it propagates up through the DOM elements as described earlier in this chapter, invoking any event handlers that have been established. Once the event reaches the context used to create the wrapped set upon which live() was called (usually the document), the context checks for elements within itself that match the live selector. The live event handlers are triggered on any elements that match, and this triggered event *doesn't* propagate.

If the logic of your page depends upon propagation and propagation order, live() might not be the best choice—especially if you mix live event handlers with native event handlers established via bind().

Secondly, the live() method can only be used on selectors, and can't be used on derived wrapped sets. For example, both of these expressions are legal:

```
$('img').live( ... )
$('img','#someParent').live( ... )
```

The first will affect all images, the second all images within the context established by #someParent. Note that when a context is specified, it must exist at the time of the call to live().

But the following expression isn't legal

```
$('img').closest('div').live( ... )
```

because it invokes live() on something other than a selector.

Even with these restrictions, live() is tremendously handy on any page with dynamic elements, but we'll see how it becomes *crucial* in pages employing Ajax in chapter 8. Later in this chapter (section 4.3), we'll see live() used extensively in a comprehensive example that employs the handy DOM manipulation methods that we learned about in chapter 3 to create dynamic elements.

REMOVING "LIVE" EVENT HANDLING

Handlers established using `live()` should be unbound with the (rather morbidly named) `die()` method, which bears a strong resemblance to its `unbind()` counterpart:

Method syntax: die

`die(eventType,listener)`

Removes live event handlers established by `live()`, and prevents the handler from being invoked on any future elements that may match the selector used for the call to `live()`.

Parameters

`eventType`	(String) If provided, specifies that only listeners established for the specified event type are to be removed.
`listener`	(Function) If provided, identifies the specific listener that's to be removed.

Returns

The wrapped set.

In addition to allowing us to manage event handling in a browser-independent manner, jQuery provides a set of methods that gives us the ability to trigger event handlers under script control. Let's look at those.

4.2.5 *Triggering event handlers*

Event handlers are designed to be invoked when browser or user activity triggers the propagation of their associated events through the DOM hierarchy. But there may be times when we want to trigger the execution of a handler under script control. We could define such event handlers as top-level functions so that we can invoke them by name, but as we've seen, defining event handlers as inline anonymous functions is much more common and so darned convenient! Moreover, calling an event handler as a function doesn't cause semantic actions or bubbling to occur.

To provide for this need, jQuery has provided methods that will automatically trigger event handlers on our behalf under script control. The most general of these methods is `trigger()`, whose syntax is as follows:

Method syntax: trigger

`trigger(eventType,data)`

Invokes any event handlers established for the passed event type for all matched elements.

Parameters

`eventType`	(String) Specifies the name of the event type for which handlers are to be invoked. This includes namespaced events. You can append the exclamation point (`!`) to the event type to prevent namespaced events from triggering.
`data`	(Any) Data to be passed to the handlers as the second parameter (after the event instance).

Returns

The wrapped set.

The `trigger()` method, as well as the convenience methods that we'll introduce in a moment, does its best to simulate the event to be triggered, including propagation through the DOM hierarchy and the execution of semantic actions.

Each handler called is passed a populated instance of `jQuery.Event`. Because there's no real event, properties that report event-specific values, such as the location of a mouse event or the key of a keyboard event, have no value. The `target` property is set to reference the element of the matched set to which the handler was bound.

Just as with actual events, triggered event propagation can be halted via a call to the `jQuery.Event` instance's `stopPropagation()` method, or a `false` value can be returned from any of the invoked handlers.

> **NOTE** The data parameter passed to the `trigger()` method is *not* the same as the one passed when a handler is established. The latter is placed into the `jQuery.Event` instance as the `data` property; the value passed to `trigger()` (and, as we're about to see, `triggerHandler()`) is passed as a parameter to the listeners. This allows both data values to be used without conflicting with each other.

For cases where we want to trigger a handler, but not cause propagation of the event and execution of semantic actions, jQuery provides the `triggerHandler()` method, which looks and acts just like `trigger()` except that no bubbling or semantic actions will occur. Additionally, no events bound by `live` will be triggered.

Method syntax: triggerHandler

`triggerHandler(eventType,data)`

Invokes any event handlers established for the passed event type for all matched elements without bubbling, semantic actions, or live events.

Parameters

`eventType`	(String) Specifies the name of the event type for which handlers are to be invoked
`data`	(Any) Data to be passed to the handlers as the second parameter (right after the event instance).

Returns

The wrapped set.

In addition to the `trigger()` and `triggerHandler()` methods, jQuery provides convenience methods for triggering most of the event types. The syntax for all these methods is exactly the same except for the method name, and that syntax is as follows:

Method syntax: *eventName*

`eventName()`

Invokes any event handlers established for the named event type for all matched elements. The supported methods are as follows:

```
blur
change
click
dblclick
error
focus
focusin
focusout
keydown
keypress
keyup
load
mousedown
mousenter
mouseleave
mousemove
mouseout
mouseover
resize
scroll
select
submit
unload
```

Parameters

`none`

Returns

The wrapped set.

In addition to binding, unbinding, and triggering event handlers, jQuery offers higher-level functions that further make dealing with events on our pages as easy as possible.

4.2.6 *Other event-related methods*

Interactive applications often employ interaction styles that are implemented using a combination of behaviors. jQuery provides a few event-related convenience methods that make it easier to use these interaction behaviors on our pages. Let's look at them.

TOGGLING LISTENERS

The first of these is the `toggle()` method, which establishes a circular progression of click event handlers that are applied on each subsequent click event. On the first click event, the first registered handler is called, on the second click, the second is called, on the third click the third is called, and so on. When the end of the list of established handlers is reached, the first handler becomes the next in line. The `toggle()` method's syntax is as follows:

Method syntax: toggle

`toggle(listener1,listener2, ...)`

Establishes the passed functions as a circular list of click event handlers on all elements of the wrapped set. The handlers are called in order on each subsequent click event.

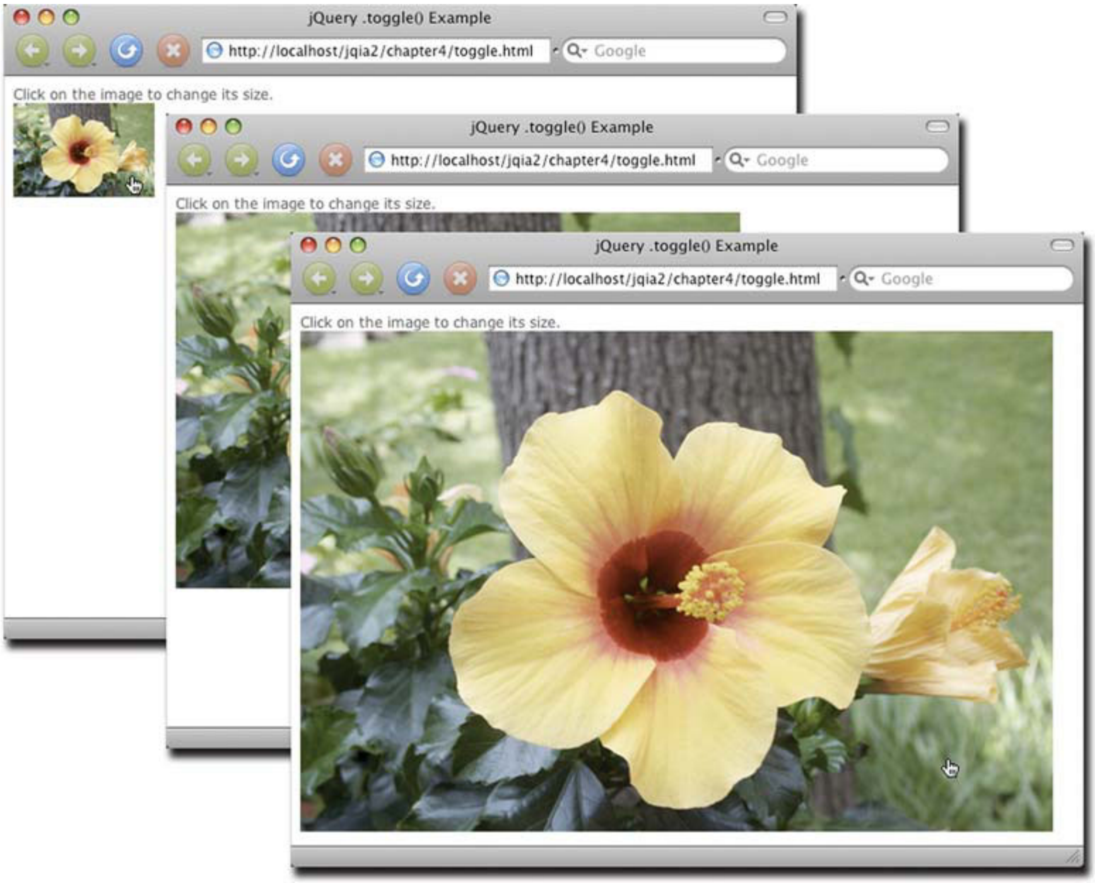
Parameters

`listenerN`	(Function) One or more functions that serve as the click event handlers for subsequent clicks.

Returns

The wrapped set.

A common use for this convenience method is to toggle the enabled state of an element back and forth on each odd or even click. For this, we'd supply two handlers; one for the odd clicks, and one for the even clicks.



Click on the image to change its size.

Figure 4.8 **jQuery's `toggle()` method lets us predefine a progression of behaviors for click events.**

But this method can also be used to create a progression through two or more clicks. Let's consider an example.

Imagine that we have a site in which we want users to be able to view images in one of three sizes: small, medium, or large. The interaction will take place through a simple series of clicks. Clicking on the image bumps it up to its next bigger size, until we reach the largest size, and it reverts back to the smallest.

Examine the progression shown in the time-lapse screenshots in figure 4.8. Each time the image is clicked, it grows to the next bigger size. If one more click were to be made, the image would revert to the smallest size.

The code for this page is shown in listing 4.6 and can be found in file chapter4/toggle.html.

Listing 4.6 Defining a progression of event handlers

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery .toggle() Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript">
      $(function(){

        $('img[src*=small]').toggle(
           function() {
             $(this).attr('src',
              $(this).attr('src').replace(/small/,'medium'));
           },
           function() {
             $(this).attr('src',
              $(this).attr('src').replace(/medium/,'large'));
           },
           function() {
             $(this).attr('src',
              $(this).attr('src').replace(/large/,'small'));
           }
        );

      });
    </script>
    <style type="text/css">
      img {
        cursor: pointer;
      }
    </style>
  </head>

  <body>

    <div>Click on the image to change its size.</div>
    <div>
      <img src="hibiscus.small.jpg" alt="Hibiscus"/>
    </div>

  </body>
</html>
```

Establishes a progression of handlers

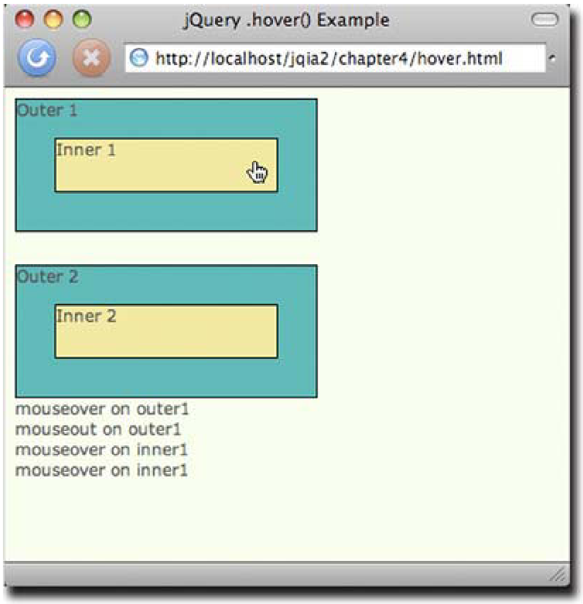
> **NOTE** If you're like your authors and pay attention to the names of things, you might wonder why this method is named `toggle()` when that really only makes sense for the case when only two handlers are established. The reason is that in earlier versions of jQuery, this method was limited to only two handlers and was later expanded to accept an arbitrary number of handlers. The name was retained for backwards compatibility.

Another common multi-event scenario that's frequently employed in interactive applications involves mousing into and out of elements.

HOVERING OVER ELEMENTS

Events that inform us when the mouse pointer has entered an area, as well as when it has left that area, are essential to building many of the user interface elements that are commonly presented to users on our pages. Among these element types, cascading menus used as navigation systems are a common example.

A vexing behavior of the mouseover and mouseout event types often hinders the easy creation of such elements: when a mouseout event fires as the mouse is moved



```
mouseover on outer1
mouseout on outer1
mouseover on inner1
mouseover on inner1
```

Figure 4.9 This page helps demonstrate when mouse events fire as the mouse pointer is moved over an area and its children.

over an area and its children. Consider the display in figure 4.9 (available in the file chapter4/hover.html).

This page displays two identical (except for naming) sets of areas: an outer area and an inner area. Load this page into your browser as you follow the rest of this section.

For the top set of rectangles on the page, the following script in the ready handler establishes handlers for the mouseover and mouseout events:

```
$('#outer1').bind('mouseover mouseout',report);
$('#inner1').bind('mouseover mouseout',report);
```

This statement establishes a function named `report` as the event handler for both the mouseover and mouseout events.

The `report()` function is defined as follows:

```
function report(event) {
  say(event.type+' on ' + event.target.id);
}
```

This listener merely adds a `<div>` element containing the name of the event that fired to a `<div>` named `console` that's defined at the bottom of the page, allowing us to see when each event fires.

Now, let's move the mouse pointer into the area labeled Outer 1 (being careful not to enter Inner 1). We'll see (from looking at the bottom of the page) that a mouseover event has fired. Move the pointer back out of the area. As expected, we'll see that a mouseout event has fired.

Let's refresh the page to start over, clearing the console.

Now, let's move the mouse pointer into Outer 1 (noting the event), but this time continue inward until the pointer enters Inner 1. As the mouse enters Inner 1, a mouseout event fires for Outer 1. If we wave our pointer to cross back and forth over the boundary between Outer 1 and Inner 1, we'll see a flurry of mouseout and mouseover events. This *is* the defined behavior, even if it's rather unintuitive. Even though the pointer is still within the bounds of Outer 1, when the pointer enters a contained element, the event model considers the transition to be leaving the outer area.

Expected or not, we don't always want that behavior. Often, we want to be informed when the pointer leaves the bounds of the outer area and don't care whether the pointer is over a contained area or not.

Luckily, some of the major browsers support a nonstandard pair of mouse events, mouseenter and mouseleave, first introduced by Microsoft in Internet Explorer. This event pair acts slightly more intuitively, not firing a mouseleave event when moving from an element to a descendant of that element. For browsers not supporting these events, jQuery emulates them so that they work the same across all browsers.

Using jQuery we could establish handlers for this set of events using the following code:

```
$(element).mouseenter(function1).mouseleave(function2);
```

But jQuery also provides a single method that makes it even easier: `hover()`. The syntax of this method is as follows:

Method syntax: hover

`hover(enterHandler,leaveHandler)`
`hover(handler)`

Establishes handlers for the mouseenter and mouseleave events for matched elements. These handlers only fire when the area covered by the elements is entered and exited, ignoring transitions to child elements.

Parameters

`enterHandler`	(Function) The function to become the mouseenter handler.
`leaveHandler`	(Function) The function to become the mouseleave handler.
`handler`	(Function) A single handler to be called for both mouseenter and mouseleave events.

Returns

The wrapped set.

We use the following script to establish mouse event handlers for the second set of areas (Outer 2 and its Inner 2 child) on the hover.html example page:

```
$('#outer2').hover(report);
```

As with the first set of areas, the `report()` function is established as both the mouseenter and mouseleave handlers for Outer 2. But unlike the first set of areas, when we pass the mouse pointer over the boundaries between Outer 2 and Inner 2, neither of these handlers (for Outer 2) is invoked. This is useful for those situations where we have no need for parent handlers to react when the mouse pointer passes over child elements.

With all these event-handling tools under our belts, let's use what we've learned so far and look at an example page that makes use of them, as well as some of the other jQuery techniques that we've learned from previous chapters!

4.3 Putting events (and more) to work

Now that we've covered how jQuery brings order to the chaos of dealing with disparate event models across browsers, let's work on an example page that puts the knowledge we've gained so far to use. This example uses not only events but also some jQuery techniques that we've explored in earlier chapters, including some heavyweight jQuery method chains. For this comprehensive example, let's pretend that we're videophiles whose collection of DVDs, numbering in the thousands, has become a huge problem. Not only has organization become an issue, making it hard to find a DVD quickly, but all those DVDs in their cases have become a storage problem—they've taken over way too much space and will get us thrown out of the house if the problem isn't solved.

We'll posit that we solved the storage side of the problem by buying DVD binders that hold one hundred DVDs each in much less space than the comparable number of DVDs in their cases. But although that saved us from having to sleep on a park bench, organizing the DVD discs is still an issue. How will we find a DVD that we're looking for without having to manually flip through each binder until we find the one we're seeking?

We can't do something like sort the DVDs in alphabetic order to help quickly locate a specific disc. That would mean that every time we buy a new DVD, we'd need to shift all the discs in perhaps dozens of binders to keep the collection sorted. Imagine the job ahead of us after we buy *Abbott and Costello Go to Mars!*

Well, we've got computers, we've got the know-how to write web applications, and we've got jQuery! So we'll solve the problem by writing a DVD database program to help keep track of what DVDs we have, and where they are.

Let's get to work!

4.3.1 Filtering large data sets

Our DVD database program is faced with the same problem facing many other applications, web-delivered or otherwise. How do we allow our users (in this case ourselves) to quickly find the information that they seek?

We could be all low-tech about it and just display a sorted list of all the titles, but that would still be painful to scroll through if there's anything more than a handful of entries. Besides, we want to learn how to do it *right* so that we can apply what we learn to real, customer-facing applications.

So no shortcuts!

Obviously, designing the entire application would be well beyond the scope of this chapter, so what we'll concentrate on is developing a control panel that will allow us to specify filters with which we can tune the list of titles that will be returned when we perform a database search.

We'll want the ability to filter on the DVD title, of course. But we'll also add the ability to filter the search based upon the year that the movie was released, its category, the binder in which we placed the disc, and even whether we've viewed the movie yet or not. (This will help answer the commonly asked question, "What should I watch tonight?")

Your initial reaction may be to wonder what the big deal is. After all, we can just put up a number of fields and be done with it, right?

Well, not so fast. A single field for something like the title is fine if, for example, we wanted to find all movies with the term "creature" in their title. But what if we want to search for either of "creature" or "monster"? Or only movies released in 1957 or 1972?

In order to provide a robust interface for specifying filters, we'll need to allow the user to specify multiple filters for either the same or different properties of the DVD. And rather than try to guess how many filters will be needed, we'll be all swank about it and create them on demand.

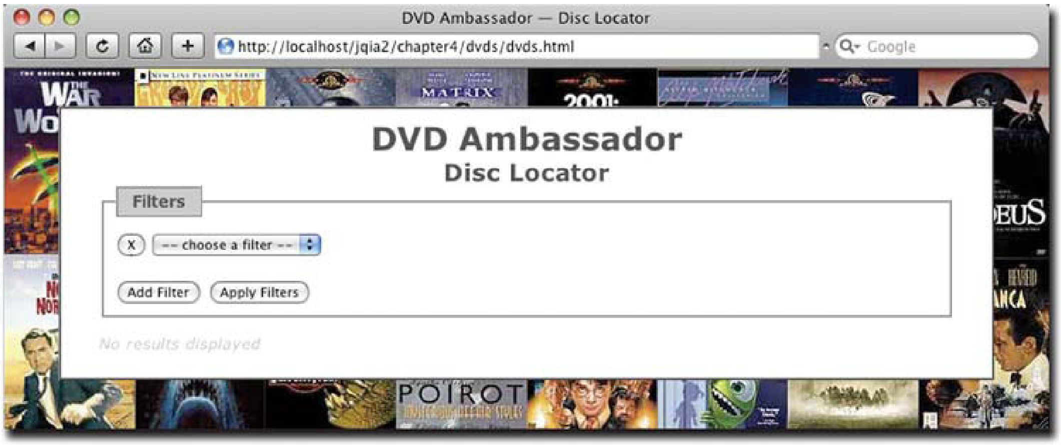
For our interface, we're going to steal a page from Apple's user interface playbook and model our interface on the way filters are specified in many Apple applications. (If you're an iTunes user, check out how Smart Playlists are created for an example.)

Each filter, one per "line," is identified by a dropdown (single-selection `<select>` element) that specifies the field that's to be filtered. Based upon the type of that field (string, date, number, and even Boolean) the appropriate controls are displayed on the line to capture information about the filter.

The user is given the ability to add as many of these filters as they like, or to remove previously specified filters.

A picture being worth a thousand words, study the time-progression display of figures 4.10a through 4.10c. They show the filter panel that we'll be building: (a) when initially displayed, (b) after a filter has been specified, and (c) after a number of filters have been specified.

As we can see by inspecting the interactions shown in figures 4.10a through 4.10c, there's going to be a lot of element creation on the fly. Let's take a few moments to discuss how we're going to go about that.



DVD Ambassador — Disc Locator

http://localhost/jqia2/chapter4/dvds/dvds.html

DVD Ambassador

Disc Locator

Filters

No results displayed

Figure 4.10a The display initially shows a single, unconfigured filter.

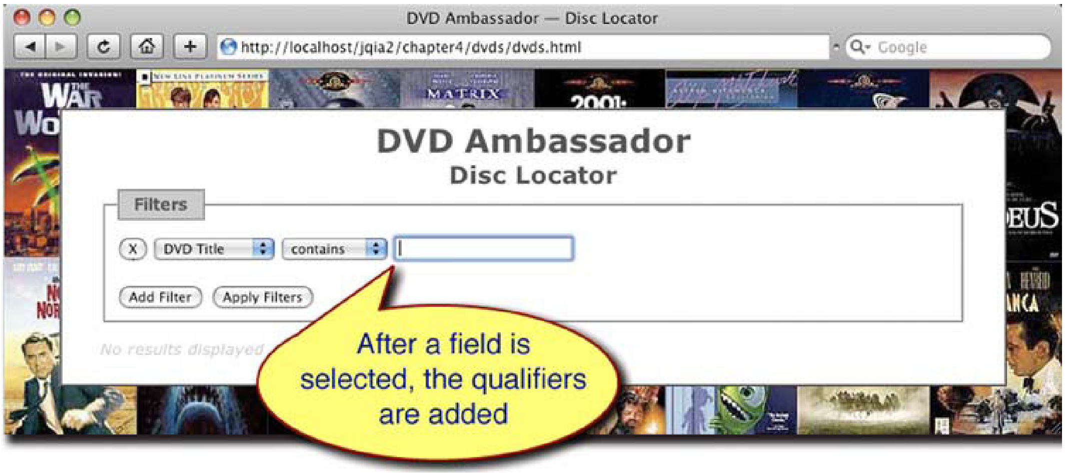
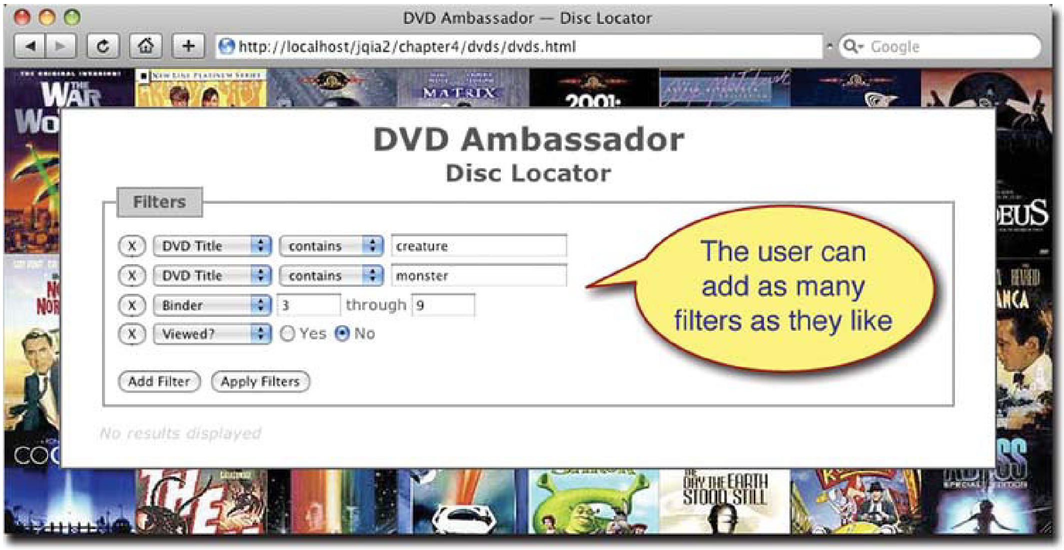

DVD Ambassador — Disc Locator
http://localhost/jqia2/chapter4/dvds/dvds.html
Google
DVD Ambassador
Disc Locator
Filters
X
DVD Title
contains
Add Filter
Apply Filters
No results displayed
After a field is selected, the qualifiers are added

Figure 4.10b After a filter type is selected, its qualifier controls are added.


DVD Ambassador — Disc Locator
http://localhost/jqia2/chapter4/dvds/dvds.html
Google
DVD Ambassador
Disc Locator
Filters
Add Filter
Apply Filters
No results displayed
The user can add as many filters as they like

X DVD Title contains creature
X DVD Title contains monster
X Binder 3 through 9
X Viewed? ○ Yes ● No

Figure 4.10c The user can add as many filters as required.

4.3.2 Element creation by template replication

We can readily see that to implement this filtering control panel, we're going to need to create a fair number of elements in response to various events. For example, we'll new to create a new filter entry whenever the user clicks the Add Filter button, and new controls that qualify that filter whenever a specific field is selected.

No problem! In the previous chapter we saw how easy jQuery makes it to dynamically create elements using the `$()` function. And although we'll do some of that in our example, we're also going to explore some higher-level alternatives.

When we're dynamically creating lots of elements, all the code necessary to create those elements and stitch together their relationships can get a bit unwieldy and a bit difficult to maintain, even with jQuery's assistance. (Without jQuery's help, it can be a complete nightmare!) What'd be great would be if we could create a "blueprint" of the complex markup using HTML, and then replicate it whenever we needed an instance of the blueprint.

Yearn, no more! The jQuery `clone()` method gives us just that ability.

The approach that we're going to take is to create sets of "template" markup that represent the HTML fragments we'd like to replicate, and use the `clone()` method whenever we need to create an instance of that template. We don't want these templates to be visible to the end user, so we'll sequester them in a `<div>` element at the end of the page that's hidden from view using CSS.

As an example, let's consider the combination of the "X" button and dropdown that identifies the filterable fields. We'll need to create an instance of this combination every time the user clicks the Add Filter button. The jQuery code to create such a button and `<select>` element, along with its child `<option>` elements, could be considered a tad long, though it would not be too onerous to write or maintain. But it'd be easy to envision that anything more complex would get unwieldy quickly.

Using our template technique, and placing the template markup for that button and dropdown in a parent `<div>` used to hide all the templates, we create markup as follows:

```
<!-- hidden templates -->
<div id="templates">
  <div class="template filterChooser">
    <button type="button" class="filterRemover" title="Remove this
     filter">X</button>

    <select name="filter" class="filterChooser" title="Select a property to
     filter">
      <option value="" data-filter-type="" selected="selected">
        -- choose a filter --</option>
      <option value="title" data-filter-type="stringMatch">DVD Title</option>
      <option value="category" data-filter-type="stringMatch">Category
      </option>
      <option value="binder" data-filter-type="numberRange">Binder</option>
      <option value="release" data-filter-type="dateRange">Release Date
      </option>
      <option value="viewed" data-filter-type="boolean">Viewed?</option>
    </select>
  </div>

  <!-- more templates go here -->

</div>
```

❶ **Encloses and hides all templates**

❷ **Defines the filterChooser template**

The outer `<div>` with `id` of `templates` serves as a container for all our templates and will be given a CSS `display` rule of `none` to prevent it from being displayed in the browser ❶.

Within this container, we define another `<div>` which we give the classes `template` and `filterChooser` ❷. We'll use the `template` class to identify templates in general, and the `filterChooser` class to identify this particular template type. We'll see how these classes are used in the code shortly—remember, classes aren't just for CSS anymore!

Also note that each `<option>` in the `<select>` has been given a custom attribute: `data-filter-type`. We'll use this value to determine what type of filter controls need to be used for the selected filter field.

Based upon which filter type is identified, we'll populate the remainder of the filter entry "line" with qualifying controls that are appropriate for the filter type.

For example, if the filter type is `stringMatch`, we'll want to display a text field into which the user can type a text search term, and a dropdown giving them options for how that term is to be applied.

We've set up the template for this set of controls as follows:

```
<div class="template stringMatch">
  <select name="stringMatchType">
    <option value="*">contains</option>
    <option value="^">starts with</option>
    <option value="$">ends with</option>
    <option value="=">is exactly</option>
  </select>
  <input type="text" name="term"/>
</div>
```

Again, we've used the `template` class to identify the element as a template, and we've flagged the element with the class `stringMatch`. We've purposely made it such that this class matches the `data-filter-type` value on the field chooser dropdown.

Replicating these templates whenever, and wherever, we want is easy using the jQuery knowledge under our belts. Let's say that we want to append a template instance to the end of an element that we have a reference to in a variable named `whatever`. We could write

```
$('div.template.filterChooser').children().clone().appendTo(whatever);
```

In this statement, we select the template container to be replicated (using those convenient classes we placed on the template markup), select the child elements of the template container (we don't want to replicate the `<div>`, just its contents), make clones of those children, and then attach them to the end of the contents of the element identified by `whatever`.

See why we keep emphasizing the power of jQuery method chains?

Inspecting the options of the `filterChooser` dropdown, we see that we have a number of other filter types defined: `numberRange`, `dateRange`, and `boolean`. So we define qualifying control templates for those filter types as well with this code:

```
<div class="template numberRange">
  <input type="text" name="numberRange1" class="numeric"/> <span>through
     </span>
  <input type="text" name="numberRange2" class="numeric"/>
</div>

<div class="template dateRange">
  <input type="text" name="dateRange1" class="dateValue"/>
  <span>through</span>
  <input type="text" name="dateRange2" class="dateValue"/>
</div>

<div class="template boolean">
  <input type="radio" name="booleanFilter" value="true" checked="checked"/>
     <span>Yes</span>
  <input type="radio" name="booleanFilter" value="false"/> <span>No</span>
</div>
```

OK. Now that we've got our replication strategy defined, let's take a look at the primary markup.

4.3.3 Setting up the mainline markup

If we refer back to figure 4.10a, we can see that the initial display of our DVD search page is pretty simple: a few headers, a first filter instance, and a couple of buttons. Let's take a look at the HTML markup that achieves that:

```
<div id="pageContent">

  <h1>DVD Ambassador</h1>
  <h2>Disc Locator</h2>

  <form id="filtersForm" action="/fetchFilteredResults" method="post">

    <fieldset id="filtersPane">
      <legend>Filters</legend>                         ❶ Container for
      <div id="filterPane"></div>                         filter instances
      <div class="buttonBar">
        <button type="button" id="addFilterButton">Add Filter</button>
        <button type="submit" id="applyFilterButton">Apply Filters</button>
      </div>
    </fieldset>

    <div id="resultsPane">                                       ❷ Container for
      <span class="none">No results displayed</span>            search results
    </div>

  </form>

</div>
```

There's nothing too surprising in that markup—or is there? Where, for example, is the markup for the initial filter dropdown? We've set up a container in which the filters will be placed ❶, but it's initially empty. Why?

Well, we're going to need to be able to populate new filters dynamically—which we'll be getting to in just a moment—so why do the work in two places? As we shall see, we'll be able to leverage the dynamic code to initially populate the first filter, so we don't need to explicitly create it in the static markup.

One other thing that should be pointed out is that we've set aside a container to receive the results ❷ (the fetching of which is beyond the scope of this chapter), and we've placed these results *inside* the form so that the results themselves can contain form controls (for sorting, paging, and so on).

OK. We have our very simple, mainline HTML laid out, and we have a handful of hidden templates that we can use to quickly generate new elements via replication. Let's finally get to writing the code that will apply the behavior to our page!

4.3.4 Adding new filters

Upon a click of the Add Filter button, we need to add a new filter to the `<div>` element that we've set up to receive it, which we've identified with the `id` of `filterPane`.

Recalling how easy it is to establish event handlers using jQuery, it should be an easy matter to add a click handler to the Add Filter button. But wait! There's something we forgot to consider!

We've already seen how we're going to replicate form controls when users add filters, and we've got a good strategy for easily creating multiple instances of these controls. But eventually, we're going to have to submit these values to the server so it can look up the filtered results in the database. And if we just keep copying the same `name` attributes over and over again for our controls, the server is just going to get a jumbled mess without knowing what qualifiers belong to which filters!

To help out the server-side code (there's a good chance we'll be writing it ourselves in any case) we're going to append a unique suffix to the `name` attribute for each filter entry. We'll keep it simple and use a counter, so that the first filter's controls will all have ".1" appended to their names, the second set ".2", and so on. That way, the server-side code can group them together by suffix when they arrive as part of the request.

> **NOTE** The ".n" suffix format was chosen for this example code because it is simple and conceptually captures what the suffix is trying to represent (groupings of parametric data). Depending upon what you are using for your server-side code, you may wish to choose alternative suffix formats that might work better in conjunction with the data-binding mechanisms that are available to you. For example, the ".n" format would not play particularly well with Java backends using property-based POJO binding mechanisms (a format of "[n]" would be better suited for this environment).

To keep track of how many filters we've added (so we can use the count as the suffix for subsequent filter names), we'll create a global variable, initialized to 0, as follows:

```
var filterCount = 0;
```

"Global variable? I thought those were evil," I hear you say.

Global variables can be a problem *when used incorrectly*. But in this case, this is truly a global value that represents a page-wide concept, and it will never create any conflicts because all aspects of the page will want to access this *single* value in a consistent fashion.

With that set up, we're ready to establish the click handler for the Add Filter button by adding the following code to a ready handler (remember, we don't want to start referencing DOM elements until after we *know* they've been created):

```
$('#addFilterButton').click(function(){
  var filterItem = $('<div>')
    .addClass('filterItem')
    .appendTo('#filterPane')
    .data('suffix','.' + (filterCount++));
  $('div.template.filterChooser')
      .children().clone().appendTo(filterItem)
      .trigger('adjustName');
});
```

❶ **Establishes click handler**

❷ **Creates filter entry block**

❸ **Replicates filter dropdown template**

❹ **Triggers custom event**

Although this compound statement may look complicated at first glance, it accomplishes a great deal without a whole lot of code. Let's break it down one step at a time.

The first thing that we do in this code is to establish a click handler on the Add Filter button ❶ by using the jQuery `click()` method. It's within the function passed to this method, which will get invoked when the button is clicked, that all the interesting stuff happens.

Because a click of the Add Filter button is going to, well, add a filter, we create a new container for the filter to reside within ❷. We give it the class `filterItem` not only for CSS styling, but to be able to locate these elements later in code. After the element is created, it's appended to the master filter container that we created with the `id` value of `filterPane`.

We also need to record the suffix that we want to add to the control names that will be placed within this container. This value will be used when it comes time to adjust the names of the controls, and this is a good example of a value that's *not* suited for a global variable. Each filter container (class `filterItem`) will have its own suffix, so trying to record this value globally would require some sort of complex array or map construct so that the various values didn't step all over each other.

Rather, we'll avoid the whole mess by recording the suffix value on the elements themselves using the very handy jQuery `data()` method. Later, when we need to know what suffix to use for a control, we'll simply look at this data value on its container and won't have to worry about getting it confused with the values recorded on other containers.

The code fragment

```
.data('suffix','.' + (filterCount++))
```

formats the suffix value using the current value of the `filterCount` variable, and then increments that value. The value is attached to the `filterItem` container using the name `suffix`, and we can later retrieve it by that name whenever we need it.

The final statement of the click handler ❸ replicates the template that we set up containing the filter dropdown using the replication approach that we discussed in the previous section. You might think that the job is over at this point, but after the cloning and appending, we execute ❹ the following fragment:

```
.trigger('adjustName')
```

The `trigger()` method is used to trigger an event handler for an event named `adjustName`.

adjustName?

If you thumb through your specifications, you'll find this event listed nowhere! It's not a standard event defined anywhere *but in this page*. What we've done with this code is to trigger a *custom event*.

A custom event is a very useful concept—we can attach code to an element as a handler for a custom event, and cause it to execute by triggering the event. The beauty of this approach, as opposed to directly calling code, is that we can register the custom handlers in advance, and by simply triggering the event, cause any registered handlers to be executed, without having to know where they've been established.

Pattern alert!

The custom event capability in jQuery is a limited example of the Observer pattern, sometimes referred to as the Publish/Subscribe pattern. We *subscribe* an element to a particular event by establishing a handler for that event on that element, and then when the event is *published* (triggered), any subscribed elements in the event hierarchy automatically have their handlers invoked. This can greatly reduce the complexity of code by reducing the *coupling* necessary.

We called this a *limited* example of the Observer pattern because subscribers are limited to elements in the publisher's ancestor hierarchy (as opposed to anywhere in the DOM).

OK, so that will *trigger* the custom event, but we need to define the handler for that event, so within the ready handler, we also establish the code to adjust the control names of the filters:

```
$('.filterItem[name]').live('adjustName',function(){
  var suffix = $(this).closest('.filterItem').data('suffix');
  if (/(\w)+\.(\d)+$/.test($(this).attr('name'))) return;
  $(this).attr('name',$(this).attr('name')+suffix);
});
```

Here we see a use of the `live()` method to proactively establish event handlers. The input elements with `name` attributes will be popping into and out of existence whenever a filter is added or removed, so we employ `live()` to automatically establish and remove the handlers as necessary. That way, we set it up *once*, and jQuery will handle the details whenever an item that matches the `.filterItem[name]` selector is created or destroyed. How easy is that?

We specify our custom event name of `adjustName` and supply a handler to be applied whenever we trigger the custom event. (Because it's a custom event, there's no possibility of it being triggered by user activity the way that, say, a click handler can be.)

Within the handler, we obtain the suffix that we recorded on the `filterItem` container—remember, within a handler, `this` refers to the element upon which the event was triggered, in this case, the element with the `name` attribute. The `closest()` method quickly locates the parent container, upon which we find the suffix value.

We don't want to adjust element names that we've already adjusted once, so we use a regular expression test to see if the name already has the suffix attached, and if so, simply return from the handler.

If the name hasn't been adjusted, we use the `attr()` method to both fetch the original name and set the adjusted name back onto the element.

At this point, it's worth reflecting on how implementing this as a custom event, and using `live()`, creates very loose coupling in our page code. This frees us from having to worry about calling the adjustment code explicitly, or establishing the custom handlers explicitly at the various points in the code when they need to be applied. This not only keeps the code cleaner, it increases the flexibility of the code.

Load this page into your browser and test the action of the Add Filter button. Note how every time you click on the Add Filter button, a new filter is added to the page. If you inspect the DOM with a JavaScript debugger (Firebug in Firefox is great for this), you'll see that the `name` of each `<select>` element has been suffixed with a per-filter suffix, as expected.

But our job isn't over yet. The dropdowns don't yet specify which field is to be filtered. When a selection is made by the user, we need to populate the filter container with the appropriate qualifiers for the filter type for that field.

4.3.5 Adding the qualifying controls

Whenever a selection is made from a filter dropdown, we need to populate the filter with the controls that are appropriate for that filter. We've made it easy for ourselves by creating markup templates that we just need to copy when we determine which one is appropriate. But there are also a few other housekeeping tasks that we need to do whenever the value of the dropdown is changed.

Let's take a look at what needs to be done when establishing the change handler for the dropdown:

```
$('select.filterChooser').live('change',function(){
  var filterType = $(':selected',this).attr('data-filter-type');
  var filterItem = $(this).closest('.filterItem');
  $('.qualifier',filterItem).remove();
  $('div.template.'+filterType)
      .children().clone().addClass('qualifier')
      .appendTo(filterItem)
      .trigger('adjustName');
  $('option[value=""]',this).remove();
});
```

❶ Establishes change handler

❷ Removes any old controls

❸ Replicates appropriate template

❹ Removes obsolete option

Once again, we've taken advantage of jQuery's `live()` method to establish a handler up front that will automatically be established at the appropriate points without further action on our part. This time, we've proactively established a change handler for any filter dropdown that comes into being ❶.

> **TIP** The ability to specify change events with `live()` is a new (and welcome) addition to jQuery 1.4.

When the change handler fires, we first collect a few pieces of information: the filter type recorded in the custom `data-filter-type` attribute, and the parent filter container.

Once we've got those values in hand, we need to remove any filter qualifier controls that might already be in the container ❷. After all, the user can change the value of the selected field many times, and we don't want to just keep adding more and more controls as we go along! We'll add the `qualifier` class to all the appropriate elements as they're created (in the next statement), so it's easy to select and remove them.

Once we're sure we have a clean slate, we replicate the template for the correct set of qualifiers ❸ by using the value we obtained from the `data-filter-type` attribute. The `qualifier` class name is added to each created element for easy selection (as we saw in the previous statement). Also note how once again we trigger the `adjustName` custom event to automatically trigger the hander that will adjust the `name` attributes of the newly created controls.

Finally, we want to remove the "choose a filter" `<option>` element from the filter dropdown ❹, because once the user has selected a specific field, it doesn't make any sense to choose that entry again. We *could* just ignore the change event that triggers when the user selects this option, but the best way to prevent a user from doing something that doesn't make sense is to not let them do it in the first place!

Once again, refer to the example page in your browser. Try adding multiple filters, and change their selections. Note how the qualifiers always match the field selection. And if you can view the DOM in a debugger, observe how the `name` attributes are augmented.

Now, for those remove buttons.

4.3.6 Removing unwanted filters and other tasks

We've given the user the ability to change the field that any filter will be applied to, but we've also given them a remove button (labeled "X") that they can use to remove a filter completely.

By this time, you should already have realized that this task will be almost trivial with the tools at our disposal. When the button is clicked, all we need to do is find the closest parent filter container, and blow it away!

```
$('button.filterRemover').live('click',function(){
  $(this).closest('div.filterItem').remove();
});
```

And yes, it turns out to be that simple.

On to a few other matters ...

You may recall that when the page is first loaded, an initial filter is displayed, even though we did not include it in the markup. We can easily realize this by simulating a click of the Add Filter button upon page load.

So, in the ready handler we simply add

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

This causes the Add Filter button handler to be invoked, just as if the user had clicked it.

And one final matter. Although it's beyond the scope of this example to deal with submitting this form to the server, we thought we'd give you a tantalizing glimpse of what's coming up in future chapters.

Go ahead and click the Apply Filters button, which you may have noted is a submit button for the form. But rather than the page reloading as you might have expected, the results appear in the `<div>` element that we assigned the `id` of `resultsPane`.

That's because we subverted the submit action of the form with a handler of our own that cancels the form submission and instead makes an Ajax call with the form contents, loading the results into the `resultsPane` container. We'll see lots more about how easy jQuery makes Ajax in chapter 8, but you might be surprised (especially if you've already done some cross-browser Ajax programming) to see that we can make the Ajax call with just one line of code:

```
$('#filtersForm').submit(function(){
  $('#resultsPane').load('applyFilters',$('#filtersForm').serializeArray());
  return false;
});
```

If you paid attention to those displayed results, you might have noted that the results are the same no matter what filters are specified. Obviously, there's no real database powering this example, so it's really just returning a hard-coded HTML page. But it's easy to imagine that the URL passed to jQuery's `load()` Ajax method could reference a dynamic PHP, Java servlet, or Rails resource that would return actual results.

That completes the page, at least as far as we wanted to take it for the purposes of this chapter, but as we know ...

4.3.7 *There's always room for improvement*

For our filter form to be considered production-quality, there's still lots of room for improvement.

Below we'll list some additional functionality that this form either requires before being deemed complete, or that would be just plain nice to have. Can you implement these additional features with the knowledge you've gained up to this point?

- Data validation is non-existent in our form. For example, the qualifying fields for the binder range should be numeric values, but we do nothing to prevent the user from entering invalid values. The same problem exists for the date fields.

  We could just punt and let the server-side code handle it—after all, it has to validate the data regardless. But that makes for a less-than-pleasant user experience, and as we've already pointed out, the best way to deal with errors is to prevent them from happening in the first place.

  Because the solution involves inspecting the `Event` instance—something that wasn't included in the example up to this point—we're going to give you the code to disallow the entry of any characters but decimal digits into the numeric fields. The operation of the code should be evident to you with the knowledge you've gained in this chapter, but if not, now would be a good time to go back and review the key points.

```
$('input.numeric').live('keypress',function(event){
  if (event.which < 48 || event.which > 57) return false;
});
```

- As mentioned, the date fields aren't validated. How would you go about ensuring that only valid dates (in whatever format you choose) are entered? It can't be done on a character-by-character basis as we did with the numeric fields.

  Later in the book, we'll see how the jQuery UI plugin solves this problem for us handily, but for now put your event-handling knowledge to the test!
- When qualifying fields are added to a filter, the user must click in one of the fields to give it focus. Not all that friendly! Add code to the example to give focus to the new controls as they are added.
- The use of a global variable to hold the filter count violates our sensibilities and limits us to one instance of the "widget" per page. Replace it by applying the `data()` method to an appropriate element, keeping in mind that we may want to use this multiple times on a page.
- Our form allows the user to specify more than one filter, but we haven't defined how these filters are applied on the server. Do they form a disjunction (in which any one filter must match), or a conjunction (in which all filters must match)? Usually, if unspecified, a disjunction is assumed. But how would you change the form to allow the user to specify which?
- What other improvements, either to the robustness of the code or the usability of the interface, would you make? How does jQuery help?

If you come up with ideas that you're proud of, be sure to visit the Manning web page for this book at http://www.manning.com/bibeault2, which contains a link to the discussion forum. You're encouraged to post your solutions for all to see and discuss!

4.4 Summary

Building upon the jQuery knowledge that we've gained so far, this chapter introduced us to the world of event handling.

We learned that there are vexing challenges to implementing event handling in web pages, but such handling is essential for creating pages in interactive web applications. Not insignificant among those challenges is the fact the there are three event models that each operate in different ways across the set of modern popularly used browsers.

The legacy Basic Event Model, also informally termed the DOM Level 0 Event Model, enjoys somewhat browser-independent operation to declare event listeners, but the implementation of the listener functions requires divergent browser-dependent code in order to deal with differences in the `Event` instance. This event model is probably the most familiar to page authors and assigns event listeners to DOM elements by assigning references to the listener functions to properties of the elements—the `onclick` property, for example.

Although simple, this model limits us to only one listener for any event type on a particular DOM element.

We can avoid this deficiency by using the DOM Level 2 Event Model, a more advanced and standardized model in which an API binds handlers to their event types and DOM elements. Versatile though this model is, it is supported only by standards-compliant browsers such as Firefox, Safari, Camino, and Opera.

For Internet Explorer, even up to IE 8, an API-based proprietary event model that provides a subset of the functionality of the DOM Level 2 Event Model is available.

Coding all event handling in a series of `if` statements—one clause for the standard browsers and one for Internet Explorer—is a good way to drive ourselves to early dementia. Luckily jQuery comes to the rescue and saves us from that fate.

jQuery provides a general `bind()` method to establish event listeners of any type on any element, as well as event-specific convenience methods such as `change()`and `click()`. These methods operate in a browser-independent fashion and normalize the `Event` instance passed to the handlers with the standard properties and methods most commonly used in event listeners.

jQuery also provides the means to remove event handlers or cause them to be triggered under script control, and it even defines some higher-level methods that make implementing common event-handling tasks as easy as possible.

As if all that were not enough, jQuery provides the `live()` method to assign handlers proactively to elements that may not even exist yet, and allows us to specify custom methods to easily register handlers to be invoked when those custom events are published.

We looked at a few examples of using events in our pages, and explored a comprehensive example that demonstrated many of the concepts that we've learned up to this point. In the next chapter, we'll look at how jQuery builds upon these capabilities to put animation and animated effects to work for us.