SAMPLE CHAPTER

Dave Crane, Bear Bibeault, and Jord Sonneveld
with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Ajax in Practice

- Get going
- Get savvy
- 60 problems solved

MANNING

Ajax in Practice

by Dave Crane
Bear Bibeault
Jord Sonneveld
with Ted Goddard, Chris Gray,
Ram Venkataraman and Joe Walker

Sample Chapter 5

brief contents

5 Handling events

This chapter covers

- Models of browser event handling
- Commonly handled event types
- Making event handling easier
- Event handling in practical applications

The days of boring HTML applications are over now that Ajax allows us to build highly interactive web applications that respond fluidly to user actions. Such user actions may include clicking a button, typing in a text box, or simply moving the mouse. User actions have been translated into events throughout the history of graphical user interfaces (GUIs), and it is no different in the browser world. When a user interacts with a web page, events are fired within the DOM hierarchy that is being interacted with, and if there are event handlers associated with the events fired on the document's elements, they will be called when the events occur. Ajax applications depend heavily on these events and their handlers; they could even be considered the lifeline of every Ajax application.

Before we get ahead of ourselves, let's see how we can add a simple event handler to a web page. In the following code snippet, notice how the `<img>` element has an `onclick` attribute. This attribute defines an event handler that will be called by the browser when the user clicks the mouse on the `<img>` element.

```
<html>
  <body>
    <img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
  </body>
</html>
```

If you load this example into a browser, you will see that when the mouse button is clicked while hovering over the image, the alert box showing the message "Woof!" is displayed, as shown in figure 5.1.

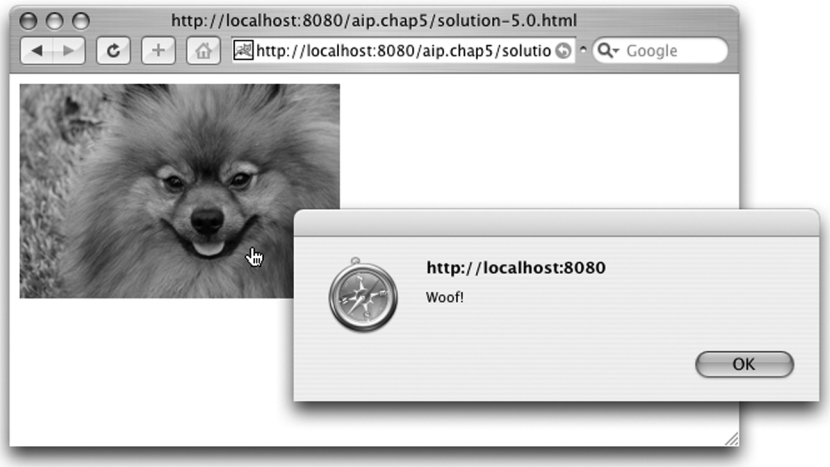

Figure 5.1 Making the dog bark

This demonstrates how easy it is to assign an event handler to a DOM element. Throughout this chapter we'll examine the major aspects of event handling. We begin by reviewing the various ways in which we can define event handlers using the various models available. We'll see how the process differs across the browser platforms and look at ways to make it portable across browser implementations. We'll also learn about the information about the event made available to event handlers when they are invoked. We'll discuss the concepts of *event bubbling* and *event capturing* that specify how events are propagated through the DOM, and we'll also look closely at the commonly handled event types. Finally, we'll whip up some real-world examples that demonstrate how we can put these concepts to use in our applications.

5.1 Event-handling models

While we've seen how easy it is to *declare* simple event handlers, you would think that writing event handlers should be just as easy. We just write some script into the handler attributes and the browser executes it when the event occurs. What could be simpler? But we wouldn't need this chapter if it were really that simple, would we?

In the present-day world, there are three event models that we need to contend with in order to use events in our web applications:

- The Basic Event Model, also informally known as the DOM Level 0 Event Model, which is fairly easy, straightforward, and reasonably cross-platform.
- The DOM Level 2 Event Model, which provides more flexibility but is supported only on standards-compliant browsers such as Firefox, Mozilla, and Safari.
- The Internet Explorer Event Model, which is functionally similar to the DOM Level 2 Model, but which is proprietary to Internet Explorer.

First we'll take a look at registering and writing handlers using the basic model, and then we'll look at using the two advanced models.

5.1.1 Basic event-handling registration

The example we examined in the chapter introduction illustrates the use of the Basic, or DOM Level 0, Model. This is the oldest approach to event handling and enjoys strong (though not complete) platform independence. It is well suited for basic event-handling needs. And as we'll see, it's not completely replaced by the more advanced models, but is typically used in conjunction with those models.

This model allows event handlers to be assigned in one of two ways:

- Inline with the HTML element markup, using event attributes of the HTML elements
- Under script control, using properties of the DOM elements

Recall the `<img>` element from our small example:

```
<img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
```

This is an example of using the *inline* technique.

The value of the `onclick` event attribute becomes the body of an anonymous function that serves as the handler for the click event. While this is easy, it has its limitations.

The best-practice design approach to building web applications separates the view of the application (HTML) from its behavior (JavaScript). Using the inline approach of defining event handlers violates this principle, and therefore it is generally recommended that use of inline handler declarations be limited or avoided.

The better approach is to attach the event handler to the DOM element under script control. This technique has become more prevalent in recent years, as the browser DOM has become more standardized and JavaScript developers have become more familiar with it. All DOM elements have properties that represent the events that can be fired on the element: for example, `onclick`, `onkeyup`, or `onchange`.

Let's rework the sample code that we saw earlier into a complete HTML document and programmatically set the `onclick` event handler of the image as shown in listing 5.1.

Listing 5.1 Assigning an event handler in script

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
      };
    </script>
  </head>
  <body>
```

❶ **Declares the page's onload handler**

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

❷ **Declares the script-free image element**

If you have downloaded the source code that accompanies this chapter from www.manning.com/crane2, you'll find this HTML document in the file `chap5/listing-5.1.html`.

While this example is functionally equivalent to our previous example, it exhibits a higher level of sophistication than the previous code. We've separated the behavior from the view by factoring the script out of the `<body>` element ❷ and into a `<script>` element in the `<head>`. Note that we have placed the code in yet another event-handler function: the `onload` event handler ❶ for the page.

Although this seems like more code to do the same thing that we saw in the first example, this technique not only improves the structure of the page but also gives us more flexibility.

An important aspect of that flexibility is the ability to control *when* handlers are established and removed. With the inline method, we're limited to establishing handlers when the page loads, and those handlers exist for the duration of the page. Assigning the handler under script control allows us to establish a handler whenever we want to. In the example of listing 5.1, we chose to establish the handler when the page loads, but we could just as easily have deferred that action until a later time as the result of some other event. Moreover, we can *remove* the event handler at any time by assigning `null` to the event property—something we can't do with inline handlers.

In our example, we created the event handler using an anonymous function literal—after all, why create a separate named function if we don't have to? But when assigning named functions as event handlers, it is important to remember not to include parentheses after the function name. We want to assign a *reference* to the function as the property value, not the result of *invoking* the function! For example, the following will invoke a function named `sayWoof()` rather than setting it as the event handler. Don't make this common mistake.

```
element.onclick = sayWoof(); //Wrong!

element.onclick = sayWoof; //Correct!
```

Although the DOM Level 0 Event Model is somewhat flexible, it does suffer from limitations; for example, it doesn't easily allow chaining of multiple JavaScript functions in response to an event.

So how would we register two functions to handle a single event? Let's initially take a rather naive approach and modify our example by adding two JavaScript event handlers to the `onclick` property of the `<img>` element, as shown in listing 5.2 (found in the file `chap5/listing-5.2.html` in the downloadable source code) with the added code highlighted in bold.

Listing 5.2 Attempting to assign two handlers

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
        document.getElementById('anImage').onclick = function() {
          alert('Woof again!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

When we run this code, it is obvious that only the second handler is called because only a single alert containing "Woof again!" is displayed. Looking at the code, this shouldn't be much of a surprise. Since `onclick` is simply a property of the `<img>` element, multiple assignments to it will overwrite any previous assignment, just as with any other property.

This poses an interesting question: is it possible to call multiple functions in response to an event? Using the DOM Level 0 Event Model, there is no means to register multiple event handlers on the same event by assigning the handlers to the element's event properties. We could factor the code from multiple functions into a single function, or we could write a function that in turn called the other functions. But each of these tactics is a rather pedestrian approach and is not very scalable. If we had no other recourse, a more sophisticated means to accomplish this would be to utilize the Observer pattern (also known as the Publisher/Subscriber pattern) in which our registered handler would serve as the observer, and other functions could register themselves as subscribers.

Luckily, we won't have to resort to such shenanigans as the browsers allow us to register multiple handlers—though, unfortunately, not in a browser-independent fashion—if we use the advanced event-handling models. Let's take a look at how to do just that.

5.1.2 Advanced event handling

In a perfect world, code written for one browser would work flawlessly in all other browsers. We don't live in that world. So when it comes to the advanced event models, we need to deal with browser differences. On the one hand, there is the World Wide Web Consortium (W3C) way of doing things, and then there is the Microsoft way of doing things. Let's look at the standardized W3C way first.

For browsers that adhere to the DOM Level 2 Event Model, a method named `addEventListener()` is defined for each DOM element and can be invoked to add an event handler to that element. This method accepts three arguments: a string declaring the event type, the event-handler function to be executed (also known as the *listener*), and a Boolean value denoting whether or not event capturing is to be enabled. We'll explain this last argument when we discuss event propagation, but for the time being, we'll just leave it set to `false`.

The event type argument expects a string containing the name of the event type to be observed. This is the attribute name for the event with the `on` prefix omitted—for example, `click` or `mouseover`.

Let's change our sample code of listing 5.2 to use this method. We'll replace the basic means (which sets the `onclick` property of the element) with calls to the `addEventListener()` method, as shown in listing 5.3 (with changes highlighted in bold).

Listing 5.3 Adding an event handler the W3C way

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof!'); },
          false);
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof again!'); },
          false);
      };
```

```
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

When *this* page is displayed and the image is clicked, both the alert boxes show up without resorting to hokey container functions to chain both event handlers. Note that this code does not work in Internet Explorer; later in this section we'll see how IE implements advanced event handling in its proprietary fashion.

Also note that, when multiple handlers for the same event on the same elements are established as we have done in our example, the DOM Level 2 Event Model does not guarantee the order in which the handlers will be executed. In testing, it was observed that the handlers seemed to be called in the order that they were established, but there is no guarantee that will always be the case and it would be folly to write code that relies on that order.

To remove an event handler from an element, we can use the `removeEventListener()` method defined for the DOM elements.

The proprietary Microsoft means of attaching events is similar in concept, but different in implementation. It uses a method named `attachEvent()` defined for the DOM elements to establish event handlers. This function accepts two arguments: the event name and the event-handler function to be executed. Unlike the event type that is used with `addEventListener()`, the event property name, complete with the `on` prefix, is expected.

Armed with this information, let's modify our sample code once again. We'll add some detection to our code and use the method that's appropriate to the containing browser. The updated code is shown in listing 5.4 (available in the downloadable source code for this chapter), once again with changes highlighted in bold.

Listing 5.4 Doing it either way

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        if (document.getElementById('anImage').attachEvent) {
          document.getElementById('anImage').attachEvent(
            'onclick',
```

```
            function() { alert('Woof!'); });
          document.getElementById('anImage').attachEvent(
            'onclick',
            function() { alert('Woof again!'); });
        }
        else {
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof!'); },
            false);
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof again!'); },
            false);
        }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

In the first line of the `onload` event handler, we check to see which method we should use. Note the use of a test known as *object detection*. Rather than testing for a specific browser, we check to see if the proprietary `attachEvent()` method exists on the element. If so, we use it; otherwise, we use the standardized W3C method.

When we display this page in any browser, it is guaranteed to work as long as the browser supports either one of these mechanisms. When we click on the image when displayed in Internet Explorer, we notice something strange: the alerts are shown in the reverse order! Or maybe not. Truth be told, as with the DOM Level 2 Event Model, we don't know in which order they will be shown. The definition of the `attachEvent()` method clearly states that multiple event handlers attached to the same event type on an element will be triggered in random order.

This completes our exploration into the ways in which event handlers can be registered across the different browsers. You saw the ease with which we can use the inline technique as well as its disadvantages. The DOM Level 0 means of registering event handlers is portable across browsers, but does not provide an automatic way of chaining multiple event-handler functions. We showed you how to attach event handlers in a more advanced way using either the DOM Level 2 or Internet Explorer models. Although this approach is flexible and allows us to dynamically attach, detach, and chain event handlers, it suffers from cross-browser issues,

forcing us to resort to object detection in order to call the method appropriate to the current browser. Fortunately, frameworks are available that abstract all these differences away and help us write code that is portable across all supported browsers. We'll see how using Prototype helps us in this manner in section 5.3.

Before we do that, let's build on our foundations of event handling in general. In the next couple of sections you'll see in detail how event information is made available to an event handler and how events are propagated through the DOM tree.

5.2 The Event object and event propagation

Two other important topics that we need to understand when dealing with events in the browser are the Event object and the manner in which events are propagated. The Event object, actually an instance of the Event class, is important for obtaining information about the event, and event propagation defines the order in which an event is delivered to its observers. First let's tackle the Event object.

5.2.1 The Event object

When an event is triggered, an instance of the Event class is created that contains a number of interesting properties describing that event. In our event handlers, we typically want to access that Event object to obtain interesting properties such as the HTML element on which the event occurred, or which mouse button was clicked (for mouse events). As with much else in the world of events, this Event object instance is made available to the event handlers in a browser-specific fashion.

For standards-compliant browsers, the Event object instance is passed as the first parameter to the event-handler function. In Internet Explorer, the instance is attached as a property to the window object instance (essentially a global variable).

Let's explore what it takes to deal with this object. Since we're getting tired of the alerts, let's also change the code to write diagnostic information into a `<div>` element below the image, as shown in listing 5.5.

Listing 5.5 Grabbing the Event instance

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick =
          function(event) {
            if (!event) event =    <-- (1) Grabs event object instance
```

```
            window.event;      ❷ Obtains event target element reference
          var target =
            event.target ? event.target : event.srcElement;
          document.getElementById('info').innerHTML +=
            'I woof at ' + target.id + '!<br/>';
        }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

In this example, we obtain a reference to the instance of Event by checking first to see if the parameter passed to the event-handler function, which we cleverly named `event`, is defined (as it will be for standards-compliant browsers) and if not, copies the `event` property from the window object ❶ where IE will have placed it.

We then want to obtain a reference to the *target element* ❷—that is, the element for which the event was generated. Again, we need to do so in a browser-specific manner as the definition of the Event class differs between IE and standard browsers.

We check to see if the standard `target` property is defined, and if not, we use the proprietary `srcElement` property.

What a pain! It seems that almost each and every step of event handling needs to do things differently in order to work in both IE and the browsers that support the W3C standards!

Well, yes, that's pretty much the case. But fear not; help is at hand. But first, let's find out what event propagation is all about.

5.2.2 Event propagation

We've focused, up to this point, on handlers that are directly defined on the elements that trigger the events, as if they are the only handlers that are significant. As it turns out, this is not the case. Rather, the event is delivered not only to the target element, but potentially to all its ancestors in the DOM tree as well. In this section, we'll see how events are propagated through the DOM tree, and learn how we can affect which event handlers are called along the way—and even how to control the propagation of an event.

We'll start by talking about how events are propagated in browsers that follow the DOM Level 2 Event Model. We'll then examine how Internet Explorer supports only a subset of that model.

In standards-compliant browsers that support the DOM Level 2 Model, when an event is triggered, that event is handled in three phases. These phases, in order, are called *capture*, *target*, and *bubble* phases.

During the capture phase, the event traverses the DOM tree from the document root element down to the target element. Any event handlers established on the traversed elements for the type of event that is being propagated are invoked *if* the event handler was registered as a *capture handler*. Remember that third parameter to the `addEventListener()` method that we've been ignoring up until now? If that parameter is set to `true`, the event handler is registered as a *capture handler*. If it's set to `false`, as we have been doing up to now, the event handler is established as a *bubble handler*. Each event handler can be either a capture or a bubble handler, but never both.

Once the event has traversed downward to the target element, activating any appropriate capture handlers along the way, the propagation enters the target phase. During this phase, the event handlers established on the target element itself are triggered as appropriate. If both a capture and a bubble handler are established on the target element, they are both invoked during this phase.

The event propagation then reverses direction and "bubbles" up the DOM tree from the target element to the root element. This is the bubble phase, and along the way, any bubble handlers established for the event type on the traversed elements are triggered.

Enough talk—how about a diagram? Let's say that we modify the body of our example program to nest the `<img>` element within two `<div>` elements as follows:

```
<div id="level1">
  <div id="level2">
    <img src="image.jpg" id="anImage"/>
  </div>
</div>
```

When we click on the image element, the click event is propagated through the DOM tree as shown in figure 5.2.

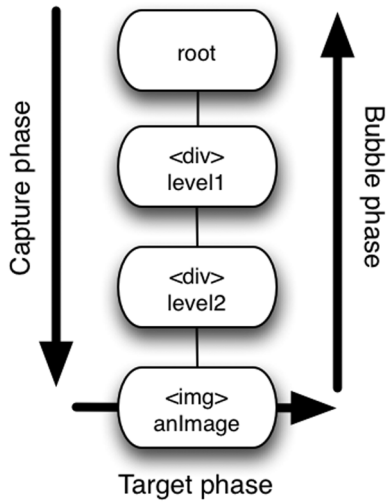

Figure 5.2 Down and up the DOM tree

Now let's see it in action. Consider the code in listing 5.6.

Listing 5.6 Establishing capture and bubble handlers

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click', react, false);
        document.getElementById('level1').addEventListener(
          'click', react, true);
        document.getElementById('level2').addEventListener(
          'click', react, false);
      }

      function react(event) {
        document.getElementById('info').innerHTML +=
          'I woof at ' + event.currentTarget.id + '!<br/>';
      }
 </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

❶ Establishes handlers

❷ Defines handler function

❸ Defines nested element

In this example, we've modified the body ❸ as described earlier, nesting the `<img>` element within two `<div>` elements.

Within the `onload` event handler ❶, we establish three event handlers: one on the `<img>` element, and one on each of the nesting `<div>` elements. Note that the event handler established on the element with the `id` of `level1` is registered as a capture handler by way of its third parameter.

All event handlers are assigned the same function, `react()` ❷, which emits a message that contains the value of the `currentTarget` property of the passed `event` instance. This property differs from the `target` property in that the `target` property identifies the element that triggered the event while `currentTarget` identifies the element that is the current subject of the event propagation—in other words, the element upon which the handler was established.

Before looking at figure 5.3, try to guess what the order of handler invocation will be. Did you get it right?

When we display this example in a standards-compliant browser (remember, the code we're using is not suited for Internet Explorer yet) and click the image, we see the display shown in figure 5.3.

The reason for the order of the output should be clear. The handler established on the `level1` element is a capture handler, while the rest are bubble handlers. The `level1` handler triggers, emitting its output, during the capture phase; the event handler on the `<img>` element triggers during the target phase; and finally, the event handler on `level2` is invoked during the bubble phase.

Internet Explorer supports only the target and bubble phases; no capture phase is supported. To modify this example for IE, we need to change the calls to the `addEventListener()` method to `attachEvent()` and alter the event-handler function as well. Unfortunately, there is no property corresponding to `currentTarget` in the Event class provided by Internet Explorer.

If you are targeting IE, and getting a reference to the current target element of the bubble phase is essential to your requirements, you'll need to come up with some underhanded means of getting a reference to that element to the event handler. One tactic that we could employ would be to use the Prototype `bind()`

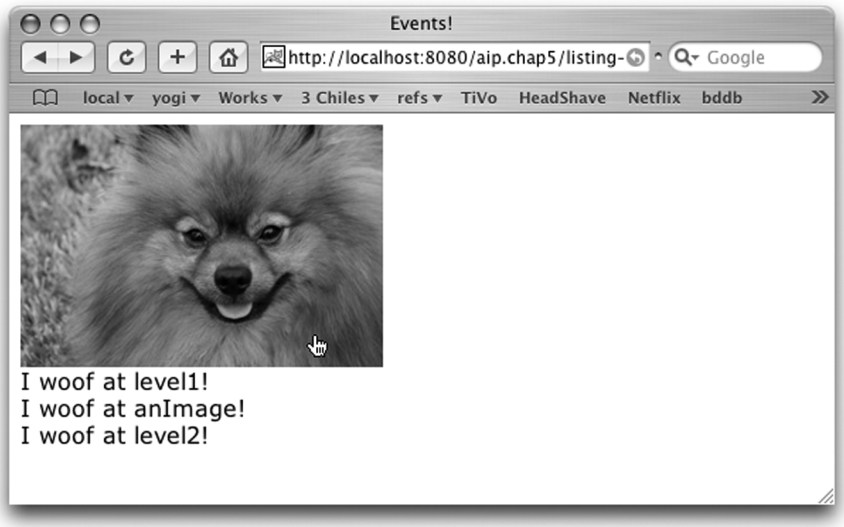

Figure 5.3 Result of capture and bubble

mechanism to force the function context object (the `this` reference) for the event handler to be the element upon which the handler is being established, as in

```
Event.observe('someId','click',someHandler.bind($('someId')));
```

Then, in the event handler, we could add

```
if (!event.currentTarget) event.currentTarget = this;
```

This would detect environments where `currentTarget` is not defined and set the context object reference into the Event instance to be used in a browser-independent fashion in the remainder of the handler. A bit Byzantine, perhaps, but useful if you absolutely must have this information available across all browsers.

Stopping propagation

There are times when you may want to prevent an event from continuing its propagation. An example is when you know that you have handled the event as much as you require and allowing the event to further propagate would trigger unwanted handlers.

In a standards-compliant browser, the `stopPropagation()` method of the Event class would be called within an event handler to prevent further propagation of the current event. In IE, the `cancelBubble` property of the Event instance is set to `true`. It may seem odd to set a property, rather than call a method, in order to effect a stop to the propagation, but that's how IE defines this action.

Preventing the default action

Some events, known as *semantic events*, trigger a default action in the browser—such as when a form is submitted, or when an anchor element is clicked.

In DOM Level 0 handlers, the value `false` can be returned in order to cause that default action to be canceled. In DOM Level 2 handlers, the `preventDefault()` method of the Event class serves the same purpose. Calling this method prevents the default action from taking place. This can be used, for example, to prevent a form from being submitted if a validation check conducted by a submit event handler determines that one or more form fields are not valid. In IE, the `returnValue` property of the Event instance is set to `false` to prevent the browser from carrying out the default action.

All these browser differences are a royal pain to deal with. Luckily, we're not the only ones who think so, and those who write JavaScript libraries have come to our aid. Let's take a look at how a now-familiar library makes event handling less painful in our pages.

5.3 Using Prototype for event handling

Several JavaScript libraries are available that simplify the process of defining event handlers by abstracting browser differences away. Prototype, which we examined previously in chapters 3 and 4 with regard to helping us write object-oriented JavaScript and make Ajax requests, also provides a simple but convenient abstraction to help us with event handling.

Prototype defines an `Event` namespace that possesses a handful of useful methods; the two most important ones are `observe()` and `stopObserving()`. The `observe()` method allows you to attach an event handler to an element, while `stopObserving()` removes event handlers from those elements.

Let's take our example of listing 5.6 and modify it using Prototype. The result is shown in listing 5.7.

Listing 5.7 Event handlers the Prototype way!

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('anImage', 'click', react, false);
        Event.observe('level1', 'click', react, true);
        Event.observe('level2', 'click', react, false);
      }

      function react(event) {
        $('info').innerHTML +=
          'I woof at ' + Event.element(event).id + '!<br/>';
      }
    </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

1 Defines event handlers

2 Declares handler function

What a difference Prototype makes! Not only were we able to use the handy `$()` function that Prototype provides, we were also able to make our example cross-browser compatible while *reducing* the amount of code we had to write.

In the `onload` event handler ❶, we used the `Event.observe()` method to establish our handlers in a cross-browser manner. We are still able to specify, for W3C-compatible browsers, whether the event handler should be a capture or a bubble handler. Under IE, this distinction will just be ignored.

In our event-handler function ❷, we used the `Event.element()` method to obtain a reference to the target element in a browser-agnostic manner.

Note that Prototype does not provide a 100 percent abstraction of the differences between browser event handling. For example, if we wanted to obtain the value of the `currentTarget` property, we'd need to do that directly, and we'd have to be sure to not make such a reference when running within IE. However, Prototype does abstract a great deal of the most commonly used event-handling requirements.

5.3.1 The Prototype Event API

This section provides a quick rundown of the API for the Prototype `Event` namespace, describing each method available.

To begin, the method

```
Event.observe(element,eventType,handler,useCapture)
```

establishes an event handler for the named event type on the passed `element`. The `useCapture` parameter may be omitted and defaults to `false`. This parameter is ignored in IE.

Next, the method

```
Event.stopObserving(element,eventType,handler,useCapture)
```

removes an event handler. The parameters should exactly match those used to establish the handler that is to be removed.

The method

```
Event.unloadCache()
```

removes all handlers established through `observe()` and frees all references in order to make them available for garbage collection. This is especially important for IE, which has a severe memory leak problem with regard to event handling. The best news is that under IE, Prototype automatically calls this method when a page is unloaded.

Next, the method

```
Event.element(event)
```

returns the target element of the passed event.

The method

```
Event.findElement(event,tagName)
```

returns the nearest ancestor of the target element for the passed event that has the passed tag name. For example, you could use this to find the nearest `<div>` parent of the target element by passing the string "div" as the `tagName` parameter.

The method

```
Event.pointerX(event)
```

returns the page-relative horizontal position of a mouse event, and the method

```
Event.pointerY(event)
```

returns the page-relative vertical position of a mouse event.

The method

```
Event.isLeftClick(event)
```

returns `true` if a mouse event was a result of a click of the primary mouse button.

Finally, the method

```
Event.stop(event)
```

stops the event from propagating any further *and* cancels any default action associated with the event.

There! That should make coding for events a lot simpler for us. Now let's turn our attention to the various event types that we commonly need to deal with.

5.4 Event types

When we consider a web application, we know that most events of interest to us occur as the result of the user interacting with the application using the mouse or the keyboard. These events are fired in the DOM element tree in response to user actions such as causing the page to load, clicking a button, moving the mouse, dragging the mouse, typing on the keyboard, or taking an action that would cause the page to unload. As we have seen, we can write event handlers for these events so that our application can respond to these actions. We'll take a closer look at the more commonly handled event types in this section, and we'll start by looking at the mouse events.

5.4.1 Mouse events

The mouse events that are most commonly handled in a web application are `mouseup`, `mousedown`, `click`, `dblclick`, and `mousemove`. When a user clicks on an element, three events are fired: `mousedown`, `mouseup`, and `click`. Let's observe this firsthand by inspecting the code in listing 5.8.

Listing 5.8 Mouse events on a single click

```
<html>
  <head>
    <title>Mouse events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('anImage', 'click', react);
        Event.observe('anImage', 'mousedown', react);
        Event.observe('anImage', 'mouseup', react);
      }

      function react(event) {
        $('info').innerHTML +=
          'I bark for ' + event.type +
          ' at (' + Event.pointerX(event) + ','+
          Event.pointerY(event) + ')!<br/>';
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

❶ Establishes mouse event handlers

❷ Emits info about event

In this code, we establish event handlers ❶ for the `click`, `mouseup`, and `mousedown` events on the `<img>` element. When the image is clicked on, the event-handler function ❷ examines the `event` instance and emits output containing the event type, as well as the page-relative coordinates of the mouse cursor at the time of the click. In the browser, we'll see the display shown in figure 5.4.

We can see from these results that when the element is clicked on, the `mousedown` event fires first, followed by `mouseup`, and finally, `click`. As an exercise, add `mousemove` or `dblclick` event handlers, and see how those events are delivered in relation to the other event types.

Figure 5.4 Reaction to mouse events

5.4.2 Keyboard events

The commonly handled keyboard events are `keyup`, `keydown`, `blur`, and `focus`. The `keyup` and `keydown` events are similar to the `mouseup` and `mousedown` events; the `keydown` event is fired when the key is pressed, and the `keyup` event is fired when the key is released.

The `focus` and `blur` events are triggered when a DOM element gains or loses focus. In any loaded page, only one DOM element can have focus at a time. The focus can be changed programmatically or as a result of user actions. When a user tabs out of a field, the `blur` event will be fired, followed by the `focus` event of the next element gaining focus. The user can also change focus by clicking on a focusable element.

Let's look at an example of how the `blur` and `focus` events work. Examine the code in listing 5.9.

Listing 5.9 Blur and focus and blur and focus and…

```
<html>
  <head>
    <title>Blur and Focus</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('nameField', 'blur', react);
```

❶ Establishes handlers on page load

```
        Event.observe('nameField', 'focus', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        $('nameField').focus();
      }

      function react(event) {
        $('info').innerHTML +=
          Event.element(event).id + ' ' +
          event.type + '<br/>';
      }
    </script>
  </head>
  <body>
    <form name="infoForm">
      <div>
        <label>Dog's name:</label>
        <input type="text" id="nameField"/>
      </div>
      <div>
        <label>Breed:</label>
        <input type="text" id="breedField"/>
      </div>
      <div>
        <label>Date of birth:</label>
        <input type="text" id="dobField"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

❷ **Assigns focus to first field**

❸ **Handles blur and focus events**

❹ **Contains focusable elements**

The structure of this example is similar to the ones that we've been looking at up to this point, but we've made some significant changes in order to shift focus from mouse events (primarily `click`) to keyboard events.

The body of the page has been modified to contain a `<form>` element ❹ in which we have defined three text fields. In the `onload` event handler ❶, we establish a `focus` event handler and a `blur` event handler for each of the text fields. We added these handlers individually for clarity. As an exercise, how would you rewrite this code so that all text fields in a form would be instrumented with the event handlers without having to list them individually?

At the conclusion of the `onload` handler, we also assign the focus ❷ to the first field in the form under script control. This is significant (besides being a friendly thing to do) because it shows us that when the page loads, the `focus` handler for that first field will trigger. This tells us that the `focus` event is triggered either when focus is assigned by script or when assigned via user activity.

This is not true for all events. The `submit` event for a form element, for example, will not be triggered when a form is submitted under script control.

We've also slightly modified our `react()` ❸ event-handler function to emit the name of the target element followed by the event type.

When this page is initially loaded into the browser, we see the display as shown in the top portion of figure 5.5. As you can see, an invocation of the `focus` event

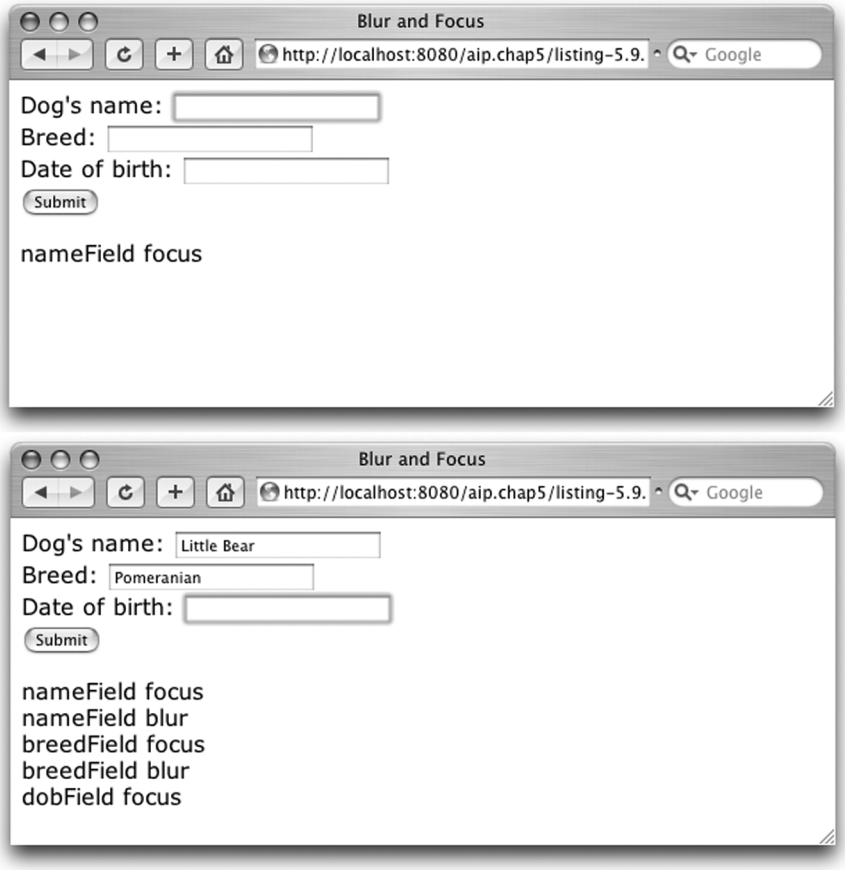

Figure 5.5 Focusing and blurring

handler has already taken place because we assigned focus to the `nameField` element in the `onload` event handler.

After filling in some data and tabbing to the `dobField` element, we can see that as we tab out of each field, the `blur` event handler is called for the element that we are leaving, and the `focus` event handler is triggered as the next element in the tab order gains focus (we'll be seeing a lot more regarding tab order in chapter 10).

Make a copy of the example code in listing 5.9 and add event handlers for the other keyboard events to text fields. Observe how they are triggered as you type the values into the fields.

5.4.3 The change event

We have seen how we can use a `blur` event handler to be notified when the user leaves an element. But it would also be useful to know whether the value of a DOM element has changed when it loses focus—for example, if we want to perform validation on a field only when its data has changed instead of every time it loses focus. For certain types of elements, such as `text`, `textarea`, `select`, and `file`, the DOM fires a `change` event when an element loses focus and the content of the element has changed between the time that field gains and loses focus.

To see this in action, we'll modify our previous example to add change event handlers to the text field elements. The result is shown in listing 5.10, with changes from listing 5.9 highlighted in bold.

Listing 5.10 Knowing what's changed

```
<html>
  <head>
    <title>Ch-ch-changes</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('nameField', 'blur', react);
        Event.observe('nameField', 'focus', react);
        Event.observe('nameField', 'change', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('breedField', 'change', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        Event.observe('dobField', 'change', react);
        $('nameField').focus();
      }
```

```
        function react(event) {
          $('info').innerHTML +=
            Event.element(event).id + ' ' +
            event.type + '<br/>';
        }
      </script>
    </head>
    <body>
      <form name="infoForm">
        <div>
          <label>Dog's name:</label>
          <input type="text" id="nameField"/>
        </div>
        <div>
          <label>Breed:</label>
          <input type="text" id="breedField"/>
        </div>
        <div>
          <label>Date of birth:</label>
          <input type="text" id="dobField"/>
        </div>
        <div>
          <input type="submit" id="submitButton"/>
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

With very little in the way of changes to the HTML document, we've added the ability to be notified when changes are effected on the text fields in our form.

If we were to load this page into our browser, enter some text into the first field, tab to the second, and then tab to the third without entering text into the second field, we'd see something like figure 5.6. As you can see, a `change` event was triggered just prior to the `blur` event for the name field, whose value was changed as a result of user input, but not for the breed field, which was not changed.

5.4.4 Page events

So far we've seen events that are fired when a user interacts with the elements within a *loaded* page, but the browser can also fire events representing page-level activity. These are called *page events*, and they occur when the document is loaded, unloaded, resized, or scrolled. Although these events sound special, we can capture them just as we do with other events by providing event handlers on the `<body>` element of the page or assigning them via the window object.

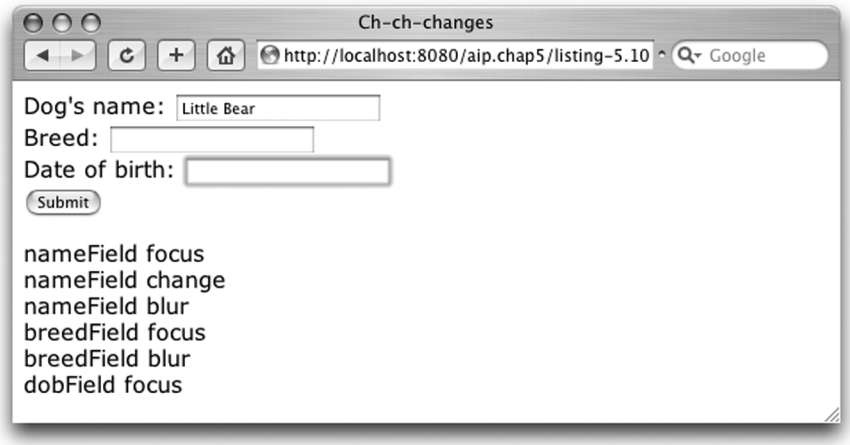

Figure 5.6 What's changed?

In every example we've examined in this chapter, we've already seen the `load` event in action; we used it to declare the other event handlers that we wanted to demonstrate. Now let's add examples of the `unload` and `onbeforeunload` events into the mix, as shown in listing 5.11.

Listing 5.11 Handling page events

```
<html>
  <head>
    <title>Page Events</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        alert('Loaded!');
        window.onunload = function() {
          alert('Unloaded!');
        }
        window.onbeforeunload =
          function() {
            return 'Leaving so soon?';
          }
        }
      }
    </script>
  </head>
  <body>
```

❶ Alerts that page is loaded

❷ Alerts that page is unloading

❸ Offers choice

```
    <a href="listing-5.11.html">Do it again!</a>
  </body>
</html>
```

As we're going to be loading and unloading the page itself, using on-page output to see what's going on won't work very well, so we've resorted to alert dialog boxes again. In the `onload` event handler, we issue an alert when the page is loaded ❶ and then proceed to establish event handlers for the `unload` and `beforeunload` events.

In the `onunload` event handler ❷, we simply issue another alert that announces that that event has triggered. But the `onbeforeunload` event handler is a bit more interesting.

In the `onunload` event handler, there's not much we can do except react to the fact that the page is unloading, but in the `onbeforeunload` event handler, we can actually affect whether or not the page will unload. If a value is returned, as in our `onbeforeunload` event handler ❸, the browser will display a dialog box that asks the user whether the page should unload. That dialog box contains the value that we returned from the handler as part of its text.

When we load this example into the browser, we get an annoying alert that announces that the page has been loaded. Upon clicking the link on the page, which we've wired to simply display the same page again, we see that the browser triggers our `onbeforeunload` event handler and, as a result of the value we returned from that handler, displays the dialog box shown in figure 5.7.

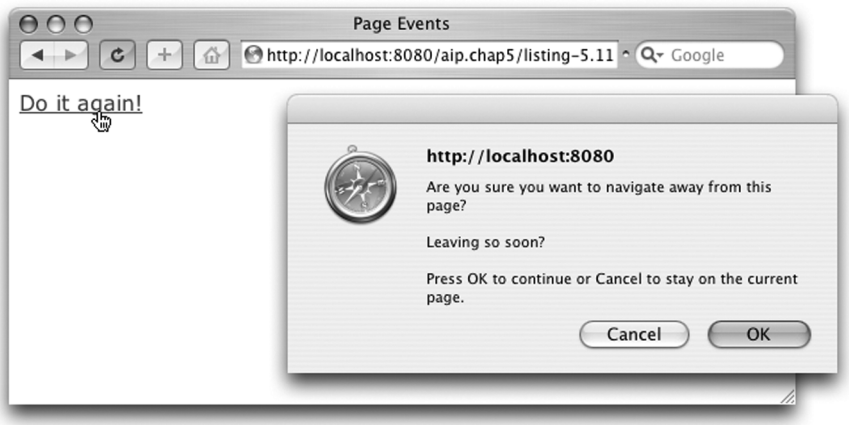

Figure 5.7 Let's chat before you go.

It doesn't take much imagination to see that this technique could be quite useful for making sure that users don't lose data when they attempt to leave a page before completing their operation. If the user clicks the Cancel button, the page navigation is canceled and the unload operation never takes place. If the user clicks the OK button, the unload operation proceeds and the user receives the alert announcing that the `unload` event handler has been called just before the page reloads.

One aside on the use of the `load` event: it's not uncommon to see pages in which a `<script>` element is placed near the bottom of the page in order to execute code as the page loads. The difference between using this tactic and implementing the `load` event is that the `load` event is guaranteed not to be triggered until after the page has *completed* loading, to include external elements such as script files, CSS style sheets, and images.

That completes our survey of event handling and our examination of some of the most commonly handled event types. Obviously, we haven't explored all events that can be fired within a web page—such an overview could take many chapters—but the information presented here is certainly enough to help you understand how event handling operates and how to handle the event types that are most typically used in modern web applications.

Now that we have a good working knowledge of event handling and the event types, let's take a look at a few practical examples of putting them to work.

5.5 Putting events into practice

The examples in this section require the services of server-side resources in order to execute. To make this as painless and simple as possible for the reader, the sample code for this chapter at www.manning.com/crane2 is already set up to be a complete and runnable web application.

If you are already running a servlet container on your system, simply create a new application context named `aip.chap5` that points to the `chap5` folder of the downloaded code as its document base.

If you are not already running a servlet engine, no need to panic. A PDF document in the `chap4` folder of the download walks you through downloading and configuring Tomcat, and also shows you how to set up application contexts.

When opening these examples in the browser, be sure to address the pages through the web server rather than merely opening the HTML pages as files. For example, to load the example in listing 5.12, you would use the address:

```
http://localhost:8080/aip.chap5/listing-5.12.html
```

This assumes, of course, that you are running the servlet container on the default port of 8080. If you've changed that port to another one, be sure to adjust the URL accordingly.

5.5.1 Validating text fields on the server

With the knowledge of how to attach `change` and `blur` event handlers to DOM elements under our belts, it is quite easy to use such handlers to validate input elements on the client to ensure that the data entered is acceptable. Simple client-side checks are easy to conduct, but sometimes business requirements dictate that the data may need to be validated using knowledge that is only available on the server. This may be because the validation is too complex to handle in JavaScript, or because the information that needs to be available in order to validate the data is too vast to send to the page for client-side use.

A common strategy used in classical web applications is to perform the simple validation on the page, and then to perform the more complex validations when the form is submitted. But with the advent of Ajax, we no longer need to put the user through this rather schizophrenic means of validation. To conduct server-assisted validation on the fly, we'll make a server request when a suitable event occurs on the client side, which will validate the data and respond to the client with an appropriate message.

We have all the information we need to solve this problem. We know that we can attach an event to a textbox to detect any changes, and that we can use that event to trigger a request to the server with Ajax. The server-side resource that such a request contacts can validate the data and send back an error message if the data proves invalid.

Note that the purpose of the example in this section is to demonstrate a real-world use of event handling, not to present a mature or sophisticated validation framework. That is a subject that *will* be discussed later in this book in chapter 6 and then again in chapter 10.

Problem

We need to validate text fields using a server-side resource when their value changes.

Solution

We've already seen how to instrument an input text element with event handlers, and this solution will do no differently. The question is: do we trap `blur` or `change` events?

The answer depends on the nature of the data and of the validations to be performed. Since we are going to be making a server round-trip whenever we want to perform a server-assisted validation operation, we want to make sure that we're not firing off requests any more than we need to.

If we know that the data is valid to begin with, we can limit ourselves to trapping `change` events. After all, there's no need to validate data that we know is already good. But in the more common case where fields may start off with unknown data (or even empty), we probably need to trap `blur` events so that the field can be validated every time it is visited.

Establishing an event handler for the field to be validated is as simple as this:

```
Event.observe('fieldId','blur',validationFunction);
```

Listing 5.12 shows a page with a small form consisting of fields for a U.S. address, city, state, and zip code. Our business requirements dictate that the zip code and address must match. This requires consulting a server-side API that the United States Postal Service (USPS) makes available and that must be consulted in the server-side code. Let's see how we handle that on the page.

Listing 5.12 Validating the zip code

```
<html>
  <head>
    <title>I Need Validation</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('zipCodeField','blur',validateZipCode);
        $('addressField').focus();
      }

      function validateZipCode(event) {
        new Ajax.Request(
          '/aip.chap5/validateZipCode',
          {
            method: 'get',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              if (transport.responseText.length != 0)
                alert(transport.responseText);
            }
          }
        );
      }
    </script>
  </head>
```

❶ Sets up event handling

❷ Initiates validation request

```
  <body>
    <form id="infoForm">    <-- ❸ Sets up data entry form
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Three major activities are addressed by this page: setting up the event handling ❶, reacting to the `blur` event by initiating the validation request to server-side resource ❷, and setting up the data entry form ❸ for the user to fill in.

In the `onload` event handler ❶ for the page, we set up the handler for the `blur` event so that the `validateZipCode()` function will be called whenever the user leaves the zip code field. This function ❷ fires off a Prototype-assisted Ajax request to a server-side resource named `validateZipCode`. As you'll see in a moment, this resource is a Java servlet that does some simplistic hand waving in order to emulate an actual zip code validation operation.

To this resource, we pass the fields of the our form utilizing the handy `serialize()` method that Prototype conveniently adds to the `<form>` element.

The server-side validation resource is defined to return an empty response if all is well and to return an error message if validation fails. So in the `onSuccess` event handler for the Ajax request, we test the text of the response and emit a simple alert if the field failed validation. Remember, more sophisticated validation handling is something that we'll explore in later chapters.

Load this page into a browser (be sure to use the web server URL, not the File menu) and fill in the fields. Note that when you leave the Zip Code field, an alert is issued displaying the validation failure message, as shown in figure 5.8.

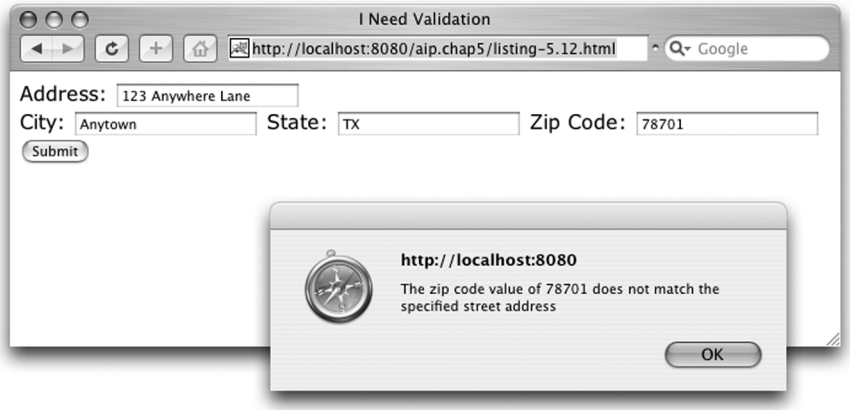

Figure 5.8 Zip code invalid!

In fact, you'll find that every zip code that you type in will generate a validation warning unless you just happened to guess the one valid zip code value of `01826`. That's because our server-side validation servlet is, of course, not really connecting to the USPS database in order to perform an actual validation. The servlet code that is faking a validation operation appears in listing 5.13.

Listing 5.13 Faking our way through a zip code validation

```
package org.aip.chap5;

import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Smoke-and-mirrors validator servlet for listing 5.12. The
 * zip code must be non-blank and equal to "01826" to be
 * considered valid.
 */
public class ZipCodeValidatorServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
    StringBuilder result = new StringBuilder ();
    String zipCodeValue = request.getParameter("zipCode");
    if (zipCodeValue.length() == 0) {
      result.append("The zip code field cannot be blank");
    }
```

```
      else if (!zipCodeValue.equals("01826")) {
        result
          .append("The zip code value of ")
          .append(zipCodeValue)
          .append(" does not match the specified street address");
      }
      response.getWriter().write(result.toString());
    }
  }
```

There's really not too much to comment on here, except that if this were an actual validation resource, all the fields for the form would be gathered, and a USPS-provided API would be utilized to perform the actual validation. Because that's not the focus of this example (or even of this book), we're just supplying a fake resource that allows us to see our client-side code in practice.

Discussion

In this section, we saw a hybrid method of using client-initiated, server-assisted validation that enables us to give users immediate feedback regarding their entered data, regardless of whether the validation needs server resources.

We used the `blur` event to detect when a user left a field in order to initiate the check. But could we be smarter about this? Once the data has been checked the first time, there's no need to go through the overhead of another server round-trip unless the data has changed. How would you modify the code to only initiate the server check if the validity of the data is unknown?

This hybrid approach of using both client-side and server-assisted on-the-fly validation is a powerful addition to our web application toolbox. Such immediate validation can prevent a lot of user frustration resulting from being told *after* the form submission that there are problems with the submitted data. So by all means, you should implement such validation. But you can never *rely* on it!

Our client-side code is readily available to anyone visiting our pages, and nefarious types will find it easy to reverse-engineer this code to submit their own false data, totally bypassing any client-side validations framework no matter how cleverly crafted. To be sure that the data is valid, *always* implement server-side validation upon form submission regardless of how much validation has been performed prior to that point. You can leverage the same code that you use for client-initiated, server-assisted validation (such as the code we examined in this example) for the final submission-time checks.

Speaking of form submission, there may be times when we want to submit a form to the server without the overhead of a complete page reload. Let's examine that next.

5.5.2 Posting form elements without a page submit

The vast majority of web pages that accept input today are written using the classical technique of submitting a form to the server when data entry is complete. This entails a complete page refresh, which may be undesirable in the context of the rich web applications that we can now deliver using Ajax.

Problem

We want to post a form to a server resource without a full-page reload.

Solution

As it turns out, the solution is almost completely trivial. In fact, we've already pretty much accomplished this task in our previous example. To "submit" the form, we'll use the same technique that we utilized in that example to send form elements to the server for validation.

Trivial and familiar as this solution might be, a few nuances make this problem worth considering. We'll take the code of our previous example, remove the validation check (so that we can focus on the submission topic), and rewire it to hijack the form-submission process in order to send the form to the server under Ajax control rather than as a normal form submission. The results are shown in listing 5.14.

Listing 5.14 Hijacking the submission process

```
<html>
  <head>
    <title>Submit!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        $('addressField').focus();
      }

      function submitMe(event) {
        new Ajax.Request(
          '/aip.chap5/handleSubmission',
          {
```

❶ Establishes submit event handler

❷ Submits form under Ajax control

```
            method: 'post',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              $('info').innerHTML = transport.responseText;
            }
          }
        );
        Event.stop(event);
      }
    </script>
  </head>

  <body>                                    ❸ Assigns normal submission action
    <form id="infoForm"
          action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

The changes to this page are subtle but significant. First, we've added a handler to the form for the `submit` event in the window's `onload` handler ❶, which will cause the `submitMe()` function to be called when the form is submitted ❷.

We'll deal with that function in just a minute, but first take a look at the change we made to the `<form>` element ❸. We added an `action` attribute that specifies a server-side resource that does not exist. By doing so, we'll quickly know if our form is ever submitted using the normal default action: the browser will display an unmistakable error page when the server reports that the resource cannot be found.

The `submitMe()` function, called when the `submit` event is triggered, initiates an Ajax request similar to the one we saw in the previous example. But in this case, we specified an HTTP method of `'post'` rather than `'get'`. The heavy lifting is done by the Prototype `serialize()` method.

The server-side resource for the request is a servlet that collects the request parameters and formats a response that contains an HTML snippet showing the names and values of those parameters. (As its operation is not germane to this discussion, we won't inspect it here. But if you're curious, you'll find the source code for the servlet in the downloadable code as the org.aip.chap5.ParameterInspectorServlet class.) This response body is displayed on the page in the `info` element.

Finally, the following statement is executed:

```
Event.stop(event);
```

This Prototype method stops the event from propagating any further and cancels the default action of the event, which in this case is the form submission. Without this statement, the form would go on submitting to the resource identified by the form's `action` attribute.

Discussion

Although this example didn't cover much new ground, it did point out some important concepts, such as using the `submit` event to prevent the submission of the form. We used an event handler and the Prototype event methods for this purpose, but if all you're trying to accomplish is preventing form submission, you can use the following form declaration to return `false` from a DOM Level 0 handler:

```
<form id="my Form" action="whatever" onsubmit="return false;">
```

In our example, we also relied heavily on the services of the Prototype `serialize()` method. This method marshals all the values of the containing form's elements and constructs either a query string or an object hash from those parameters. Because we specified `true` as the parameter to this method, it returns an object hash, which is the preferred technique for Prototype 1.5.

When this page is loaded, data entered, and the Submit button clicked (or the Enter key pressed), the display appears as shown in figure 5.9.

That was all pretty easy. But what if we want to be slightly pickier?

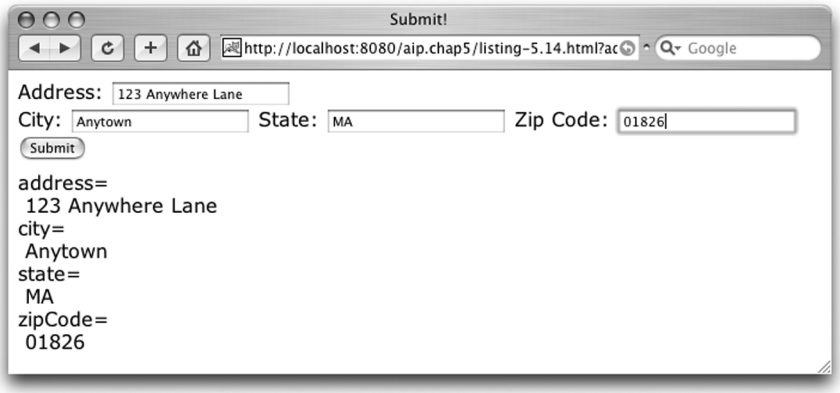

Figure 5.9 Submitting without submitting!

5.5.3 Submitting only changed elements

The previous example showed us that we can take control of the form-submission process and use event handling to reroute the submitted data to an Ajax request. Prototype's `serialize()` method made it almost trivial for us to gather all the data elements of a form to send to the server.

But what if we don't want to send *all* the form data? What if we only want to send data elements that have changed? Indeed, why make the request at all if none of the data has changed? We could use the `change` event of the form elements to know when an element's value has changed, but how do we best keep track of this information for use when it comes time to send the data to the server?

We could be sophomoric about it and store the information in global variables. But not only would that be inelegant, it would also create severe problems on pages with multiple forms, and is not an object-oriented approach.

We could be sophisticated about it and store the information right on the element itself by adding a custom property, as follows:

```
element.hasChanged = true;
```

We could then loop through the elements when it comes time to gather the data for submission, looking for elements that have this property set.

Or better yet, we can be clever about it (that sounds so much better than lazy) and leverage code that we already have handy. Listing 5.15 shows just such an approach.

Listing 5.15 Submitting only changed data

```
<html>
  <head>
    <title>Submit, or not!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        Event.observe('infoForm','change',
                      markChanged);
        $('addressField').focus();
      }

      function markChanged(event) {
        Event.element(event).addClassName('changedField');
      }

      function submitMe(event) {
        var changedElements = $$('.changedField');
        if (changedElements.length > 0 ) {
          var parameters = {};
          changedElements.each(
            function(element) {
              parameters[element.name] = element.value;
              element.removeClassName('changedField');
            }
          );
          new Ajax.Request(
            '/aip.chap5/handleSubmission',
            {
              method: 'post',
              parameters: parameters,
              onSuccess: function (transport) {
                $('info').innerHTML = transport.responseText;
              }
            }
          );
        }
        Event.stop(event);
      }
    </script>
  </head>

  <body>
    <form id="infoForm" action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
```

❶ Establishes change handler on form

❷ Marks target element as changed

❸ Collects only changed elements

```
          <label>City:</label>
          <input type="text" id="cityField" name="city"/>
          <label>State:</label>
          <input type="text" id="stateField" name="state"/>
          <label>Zip Code:</label>
          <input type="text" id="zipCodeField" name="zipCode"/>
        </div>
        <div>
          <input type="submit" id="submitButton"/>
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

In this example we've made some minor but significant changes to the code in listing 5.14. In the `onload` event handler, we've established a `change` event handler on the form ❶. We could have looped through the form, adding a handler on each individual element, but why bother when the form will receive the event notification during the bubble phase?

The handler function, `markChanged()` ❷, which will be called whenever a form element has changed, obtains a reference to the event's target element and adds the CSS class `changedField` to that element.

Huh? What does CSS have to do with keeping track of changed fields? All is revealed when we examine the changes to the `submitMe()` event-handler function.

In that function ❸, we use the Prototype `$$()` function. This handy function returns an array of all elements that match the CSS selector passed as its parameter. Since we specified the string `'.changedField'`, an array of all elements marked with that CSS class name is returned.

If that array is empty, we simply skip over the code that submits the request. Otherwise, we loop through the elements, creating an object hash of the name/value pairs that we gather from the array elements. That hash is then used as the parameter set for the Ajax request.

Since the data has been submitted and is no longer considered changed, we remove the CSS class name `changedField` from the elements, and we're good to go again!

Discussion

This example builds on the code in listing 5.14 to limit the parameters submitted on the Ajax request to those that have changed value, and to completely skip submitting the request if no changes have taken place.

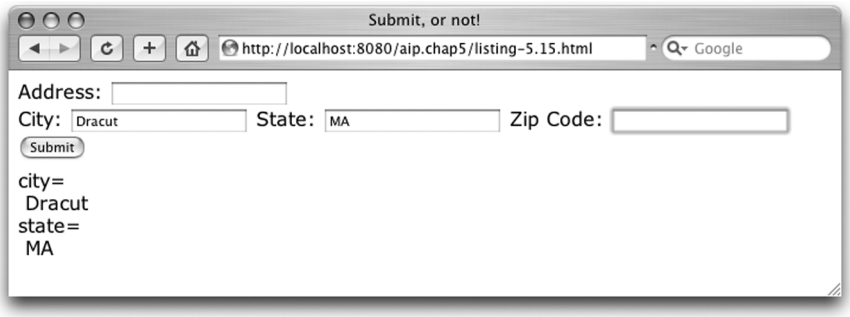

Figure 5.10 Submitting only what counts

We used a `change` event handler on the form to catch changes to all its elements, cleverly taking advantage of the bubble phase of event propagation. And we saw a clever way of marking elements for later identification through the use of CSS class names and the Prototype `$$()` function.

When displayed in the browser, and with only the City and State fields changed, we see the display as shown in figure 5.10.

5.6 Summary

In this chapter, we saw some interesting and powerful techniques to add interactivity to web applications. We looked at the various ways in which you can add event handlers to a DOM element, and we saw how the Prototype JavaScript library greatly simplifies the process of attaching and writing event handlers. We looked at all the major event types, and we examined many code snippets that demonstrated how these events can be used in our web applications. We also looked at some validation and form submission examples, something we'll cover more in-depth in the next chapter.

WEB PROGRAMMING

Ajax IN PRACTICE

Dave Crane, Bear Bibeault, and Jord Sonneveld
with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Collectively, web developers have learned a huge amount about Ajax. But it's difficult for any one of them to access and distill all that knowledge. Fortunately, it's now unnecessary: this book collects the most valuable techniques developed by the Ajax community and puts them in your hands.

Ajax in Practice gives you 60 best-practice Ajax techniques illustrated with crisp examples and tons of well-explained code you can reuse. All this is presented in an easy-to-follow, repeating format. The book starts by covering the prerequisites—key Ajax frameworks and object-oriented JavaScript (something you'll need if you want to write scalable Ajax code). Then, it helps you master practical methods for event handling, validation, and state management. A thorough discussion makes each example clear and shows how individual techniques can be combined and extended.

"A 'second-generation' book that distills experience-based practices. Confident and balanced!"

—Ernest J. Friedman-Hill
Sandia National Laboratory
Author of *Jess in Action*

"Any Ajax coder will benefit. [This book] will be useful for years to come."

—Curt Christianson
Microsoft MVP

You'll learn how to

- Implement drag and drop the right way (Chapter 9)
- Control the propagation of an event through the DOM tree (Chapter 5)
- Add back-button and undo support (Chapter 8)
- Implement effective navigation strategies (Chapter 7)
- Prefetch data to improve performance (Chapter 11)
- Build a Yahoo! Maps + Flickr mashup (Chapter 13)

Ajax in Practice brings together a team of experts including **Dave Crane**, leading Ajax authority and best-selling author of Manning's *Ajax in Action*, **Bear Bibeault** of Works.com and JavaRanch, **Jord Sonneveld** of Google, **Chris Gray** of Infor, **Ram Venkataraman** of JBoss, **Ted Goddard** of IceFaces, and **Joe Walker**, creator of DWR.

For more information, code samples, and to purchase an ebook
visit www.manning.com/AjaxinPractice

ISBN-10: 1-932394-99-0
ISBN-13: 978-1-932394-99-3

$44.99/Can$ 58.99