CHAPTER 6

Events

The most important aspect to unobtrusive DOM scripting is the use of dynamically bound *events*. The ultimate goal of writing usable JavaScript code is to have a web page that will work for the users, no matter what browser they're using or what platform they're on. To accomplish this, you set a goal of the features that you want to use, and exclude any browsers that do not support them. For the unsupported browsers, you then give them a functional, albeit less interactive, version of the site. The benefits to writing JavaScript and HTML interactions in this manner include cleaner code, more accessible web pages, and better user interactions. All of this is accomplished by using DOM events to improve the interaction that occurs in web applications.

The concept of events in JavaScript has advanced through the years—to the reliable, semiusable plateau where we now stand. Thankfully, due to the general similarities that exist, you can develop some excellent tools to help you build powerful, cleanly written web applications.

In this chapter I'm going to start with an introduction to how events work in JavaScript and how it compares to event models in other languages. Then you're going to look at what information the event model provides you with and how you can best control it. After looking at binding events to DOM elements and the different types of events that are available, I conclude by showing how to integrate some effective unobtrusive scripting techniques into any web page.

Introduction to JavaScript Events

If you look at the core of any JavaScript code, you'll see that events are the glue that holds everything together. In a nicely designed JavaScript application, you're going to have your data source and its visual representation (inside of the HTML DOM). In order to synchronize these two aspects, you're going to have to look for user interactions and attempt to update your web site accordingly. The combination of using the DOM and JavaScript events is the fundamental union that makes all modern web applications what they are.

Asynchronous Events vs. Threads

The event system in JavaScript is rather unique. It operates completely asynchronously using no threads at all. This means that all code in your application will be reliant upon other actions—such as a user's click or a page loading—triggering your code.

The fundamental difference between threaded program design and asynchronous program design is in how you wait for things to happen. In a threaded program you would keep checking over and over whether your condition has been met. Whereas in an asynchronous program you would simply register a callback function with an event handler, and then whenever that event occurs, the handler would let you know by executing your callback function. Let's explore how a JavaScript program could be written if it used threads, and how a JavaScript program is written using asynchronous callbacks.

JavaScript Threads

As it stands today, JavaScript threads do not exist. The closest that you can get is by using a setTimeout() callback, but even then, it's less than ideal. If JavaScript were a traditional threaded programming language, something like the code shown in Listing 6-1 would work. It is a mock piece of code in which you're waiting until the page has completely loaded. If JavaScript were a threaded programming language, you would have to do something like this. Thankfully, that is not the case.

Listing 6-1. *Mock JavaScript Code for Simulating a Thread*

```
// NOTE: This code DOES NOT work!
// Wait until the page is loaded, checking constantly
while ( ! window.loaded() ) { }

// The page is loaded now, so start doing stuff
document.getElementById("body").style.border = "1px solid #000";
```

If you'll notice, in this code there is a loop that's continually checking to see if window.loaded() returns true or not. Regardless of the fact that there's no loaded() function on the window object, having a loop like that doesn't work in JavaScript. This is due to the fact that all loops in JavaScript are blocking (this means that nothing else can happen until they finish running). If JavaScript were able to handle threads, you would see something like Figure 6-1. In the figure, the while loop in your code continually checks to see if the window is loaded. This does not work in JavaScript due to the fact that all loops are blocking (in that no other operations can be executed while the loop is operating).

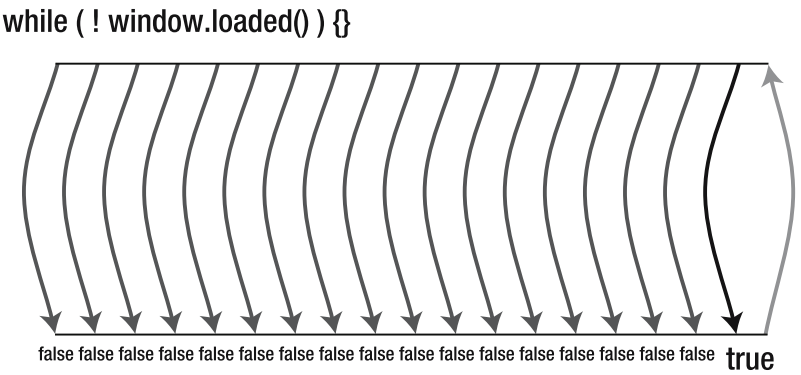

Figure 6-1. *What you'd see if JavaScript were able to handle threads*

In reality, since our while loop continues running and blocking the normal flow of the application, it'll never reach a true value. The result is that the user's browser will hang and stall and possibly crash. The lesson that you can take away from this is that if you ever see anyone claiming that using a while loop to wait for an action works (in JavaScript), they're probably lying or very confused.

Asynchronous Callbacks

The programmatic alternative to using threads to constantly check for updates is to use asynchronous callbacks, which is what JavaScript uses. Using plain terminology, you tell a DOM element that anytime an event of a specific type is called, you want a function to be called to handle it. This means that you can provide a reference to the code that you wish to be executed when needed and the browser takes care of all the details. A sample piece of code using event handlers and callbacks is shown in Listing 6-2. You see the actual code required to bind a function to an event handler (window.onload) in JavaScript. window.onload() will be called whenever the page has been loaded. This is also the case for other common events such as click, mousemove, and submit.

Listing 6-2. *Asynchronous Callbacks in JavaScript*

```
// Register a function to be called whenever the page is loaded
window.onload = loaded;

// The function to call whenever the page is loaded.
function loaded() {
    // The page is loaded now, so start doing stuff
    document.getElementById("body").style.border = "1px solid #000";
}
```

Comparing the code in Listing 6-2 to the code shown in Listing 6-1, you see a distinct difference. The only code that is executed right away is the binding of the event handler (the loaded function) to the event listener (the onload property). The browser, whenever the page is completely loaded, calls the function associated with window.onload and executes it. The flow of the JavaScript code looks something like what's shown in Figure 6-2. The figure shows a representation of using callbacks to wait for the page to load in JavaScript. Since it's actually impossible to wait for something, you register a callback (loaded) with a handler (window.onload), which will be called whenever the page is fully loaded.

One point that isn't immediately apparent with our simple event listener and handler is that the order of events can vary and can be handled differently depending on the type of event and where in the DOM the element exists. We'll look at the two different phases of events in the next section and what makes them so different.

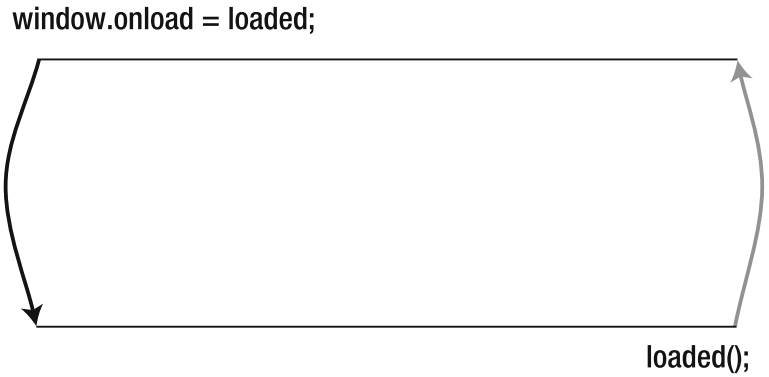

Figure 6-2. *A representation of using callbacks to wait for the page to load*

Event Phases

JavaScript events are executed in two phases called the *capturing* and *bubbling* phases. What this means is that when an event is fired from an element (e.g., the user clicking a link causing the click event to fire), the elements that are allowed to handle it, and in what order, vary. You can see an example of the execution order in Figure 6-3. The figure shows what event handlers are fired, in what order, whenever a user clicks the first <a> element on the page.

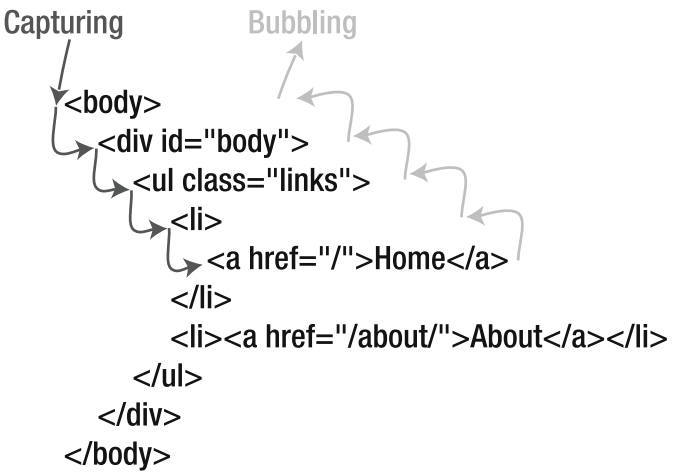

Figure 6-3. *The two phases of event handling*

Looking at a simple example of someone clicking a link (in Figure 6-3), you can see the order of execution for an event. Pretending that the user clicked the <a> element, the click handler for the document is fired first, then the <body>'s handler, then the 's handler, and so on, down to the <a> element; this is called the *capturing* phase. Once that finishes, it

moves back up the tree again, and the <li>, <ul>, , <body>, and document event handlers are all fired, in that order.

There are very specific reasons why event handling is built this way, and it works very well. Let's look at a simple example. Say you want each of the <li> elements to change its background color whenever a user moves his mouse over them, and change back again when the mouse moves off—a common need for most menus. The code shown in Listing 6-3 does exactly this.

Listing 6-3. *A Tabbed-Navigation Scenario with Hovering Effects*

```
// Find all the <li> elements, to attach the event handlers to them
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

    // Attach a mouseover event handler to the <li> element,
    // which changes the <li>s background to blue.
    li[i].onmouseover = function() {
        this.style.backgroundColor = 'blue';
    };

    // Attach a mouseout event handler to the <li> element
    // which changes the <li>s background back to its default white
    li[i].onmouseout = function() {
        this.style.backgroundColor = 'white';
    };

}
```

This code behaves exactly as you'd imagine: you mouse over an <li> element and its background color is changed; you move your mouse off of it, and the color goes back. However, what you don't realize is that you're actually toggling two different elements every time you move your mouse over the <li>. Since the <li> element also contains an <a> element, you're moving your mouse over it, instead of just the <li>. Let's look at the exact flow of the event calls:

1. * mouseover*: You move your mouse over the <li> element.
2. * mouseout*: You move from the <li> to the <a> contained inside of it.
3. *<a> mouseover*: Your mouse is now over the <a> element.
4. * mouseover*: The <a> mouseover event bubbles up to the <li> mouseover.

You may notice from the way that you're calling the events, that you're completely ignoring the capturing event phase; don't worry, I haven't forgotten about it. The way that you're binding the event listeners is by using an old "traditional" means of binding events by setting the onevent property of an element, which only supports event bubbling, not capturing. This way of event binding, and others, is discussed in the next section.

In addition to the strange order of event calls, you may have noticed two unexpected actions: the mouseout of the <li> element and the <a> to <li> mouseover bubbling. Let's look at those in detail.

The first mouseout event occurs because, as far as the browser is concerned, you've left the realm of the parent <li> element and have moved into another element. This is due to the fact that whichever element is currently on top of the elements beneath them (as the <a> element is to its <li> parent) receives the immediate focus of the mouse.

The <a> mouseover bubbling to the parent <li> element ends up becoming our saving grace in this piece of code. Since you haven't actually bound any sort of listener to the <a> element, the event simply continues on up the DOM tree, looking for another element that is listening. The first element that it encounters in its bubbling process is the <li> element, which is listening for incoming mouseover events (and which is exactly what you want).

One point that you should consider is, what if you did bind an event handler to the <a> element's mouseover event? Is there any way that you could stop the bubbling of the event? This is an important and useful topic that I will be covering next.

Common Event Features

A great aspect of JavaScript events is that they have a number of relatively consistent features that give you more power and control when developing. The simplest and oldest concept is that of the event object, which provides you with a set of metadata and contextual functions to allow you to deal with things such as mouse events and keyboard presses. Additionally, there are functions that can be used to modify the normal capture/bubbling flow of an event. Learning these features inside and out can make your life much simpler.

The Event Object

One standard feature of event handlers is some way to access an event object, which contains contextual information about the current event. This object serves as a very valuable resource for certain events. For example, when handling keyboard presses you can access the keyCode property of the object to get the specific key that is pressed. More details concerning the specifics of the event object can be found in Appendix B.

The tricky part of the event object, however, is that Internet Explorer's implementation is different from the W3C's specification. Internet Explorer has a single global event object (which can be reliably found in the global variable property window.event), whereas every other browser has a single argument passed to it, containing the event object. An example of reliably using the event object is shown in Listing 6-4. The listing is an example of modifying a common <textarea> element to behave differently. Typically, users can hit the Enter key inside of a textarea, causing there to be extra end lines. But what if you don't want that and instead only want a large text box? This function provides just that.

Listing 6-4. *Overriding Functionality Using DOM Events*

```
// Find the first <textarea> on the page and bind a keypress listener
document.getElementsByTagName("textarea")[0].onkeypress = function(e){
    // If no event object exists, then grab the global (IE-only) one
    e = e || window.event;

    // If the Enter key is pressed, return false (causing it to do nothing)
    return e.keyCode != 13;
};
```

There are a lot of attributes and functions contained within the event object, and what they're named or how they behave varies from browser to browser. I won't go into the particulars right now, but I highly recommend that you read Appendix B, which has a large list of all the event object features, how to use them, and examples of them in use.

The this Keyword

The this keyword (as discussed in Chapter 2) serves as a way to access the current object within the scope of a function. Modern browsers give all event handlers some context using the this keyword. As usual, only some of them (and only some methods) play nice and set it equal to the current element; this will be discussed in depth in a minute. For example, in Listing 6-5, I can take advantage of this fact by only creating one generic function for handling clicks but using the this keyword to determine which element is currently being affected. The listing shows an example of using only one function to handle a click event, but since it uses the this keyword to reference the element, it will work as intended.

Listing 6-5. *Changing the Background and Foreground Color of All <li> Elements Whenever They Are Clicked*

```
// Find all <li> elements and bind the click handler to each of them
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {
    li[i].onclick = handleClick;
}

// The click handler - when called it changes the background and
// foreground color of  the specified element
function handleClick() {
    this.style.backgroundColor = "blue";
    this.style.color = "white";
}
```

The this keyword really is nothing more than a convenience, however, I think you'll find that it can greatly reduce the complexity of your JavaScript code when using it properly. I try to write all the event-related code in this book using the this keyword.

Canceling Event Bubbling

Since you know how event capturing/bubbling works, let's explore how you can take control of it. An important point brought up in the previous example is that if you want an event to only occur on its target and not its parent elements, you have no way to stop it. Stopping the flow of an event bubble would cause an occurrence similar to what is shown in Figure 6-4, which shows the result of an event being captured by the first <a> element and the subsequent bubbling being canceled.

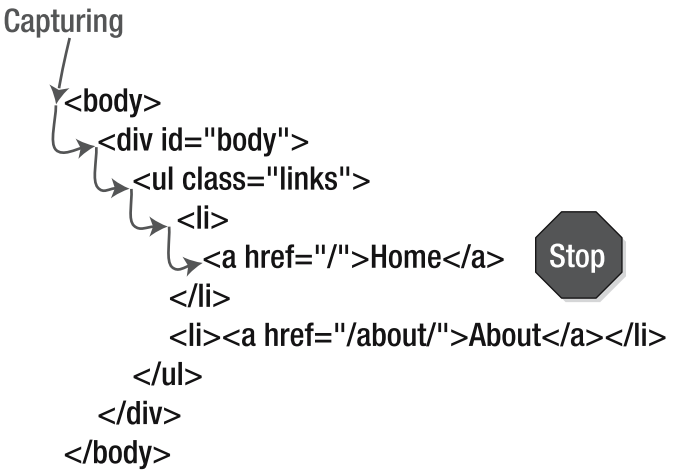

Figure 6-4. *The result of an event being captured by the first <a> element*

Stopping the bubbling (or capturing) of an event can prove immensely useful in complex applications. Unfortunately, Internet Explorer offers a different way than all other browsers to stop an event from bubbling. A generic function to cancel event bubbling can be found in Listing 6-6. The function takes a single argument: the event object passed into an event handler. The function handles the two different ways of canceling the event bubbling: the standard W3C way, and the nonstandard Internet Explorer way.

Listing 6-6. *A Generic Function for Stopping Event Bubbling*

```
function stopBubble(e) {
    // If an event object is provided, then this is a non-IE browser
    if ( e && e.stopPropagation )
        // and therefore it supports the W3C stopPropagation() method
        e.stopPropagation();
    else
        // Otherwise, we need to use the Internet Explorer
        // way of cancelling event bubbling
        window.event.cancelBubble = true;
}
```

What you're probably wondering now is, when would I want to stop the bubble of events? Honestly, the majority of the time you'll probably never have to worry about it. The need for it begins to arise when you start developing dynamic applications (especially ones that deal with the keyboard or mouse).

Listing 6-7 shows a brief snippet that adds a red border around the current element that you're hovered over. You do this by adding a mouseover and a mouseout event handler to every DOM element. If you don't stop the event bubbling, every time you move your mouse over an element, the element and all of its parent elements will have the red border, which isn't what you want.

Listing 6-7. *Using stopBubble() to Create an Interactive Set of Elements*

```
// Locate, and traverse, all the elements in the DOM
var all = document.getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {

    // Watch for when the user moves his mouse over the element
    // and add a red border around the element
    all[i].onmouseover = function(e) {
        this.style.border = "1px solid red";
        stopBubble( e );
    };

    // Watch for when the user moves back out of the element
    // and remove the border that we added
    all[i].onmouseout = function(e) {
        this.style.border = "0px";
        stopBubble( e );
    };

}
```

With the ability to stop the event bubbling, you now have complete control over which elements get to see and handle an event. This is a fundamental tool necessary for exploring the development of dynamic web applications. The final aspect is to cancel the default action of the browser, allowing you to completely override what the browser does and implement new functionality instead.

Overriding the Browser's Default Action

For most events that take place, the browser has some default action that will always occur. For example, clicking an <a> element will take you to its associated web page; this is a default action in the browser. This action will always occur after both the capturing and the bubbling event phases, as shown in Figure 6-5. This particular example shows the results of a user clicking an <a> element in a web page. The event begins by traveling through the DOM in both a capturing and bubbling phase (as discussed previously). However, once the event has finished traversing, the browser attempts to execute the default action for that event and element. In this case, it's visiting the / web page.

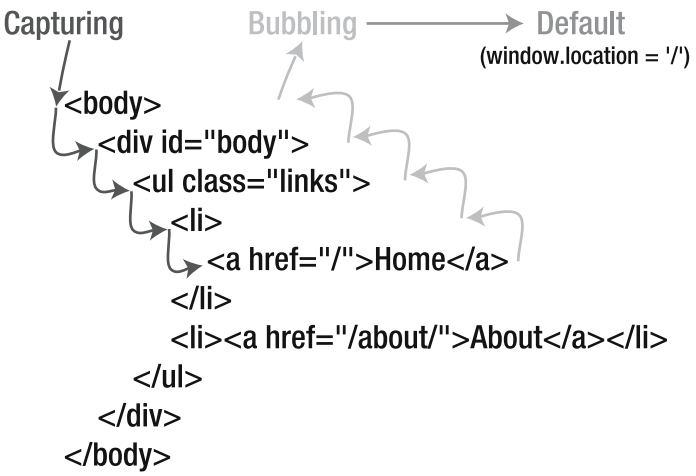

Figure 6-5. *The full life cycle of an event*

Default actions can be summarized as anything that the browser does that you do not explicitly tell it to do. Here's a sampling of the different types of default actions that occur, and on what events:

- Clicking an <a> element will redirect you to a URL provided in its href attribute.
- Using your keyboard and pressing Ctrl+S, the browser will attempt to save a physical representation of the site.
- Submitting an HTML <form> will submit the query data to the specified URL and redirect the browser to that location.
- Moving your mouse over an <img> with an alt or a title attribute (depending on the browser) will cause a tool tip to appear, providing a description of the <img>.

All of the previous actions are executed by the browser even if you stop the event bubbling or if you have no event handler bound at all. This can lead to significant problems in your scripts. What if you want your submitted forms to behave differently? Or what if you want <a> elements to behave differently than their intended purpose? Since canceling event bubbling isn't enough to prevent the default action, you need some specific code to handle that directly. As with canceling event bubbling, there are two ways of stopping the default action from occurring: the IE-specific way and the W3C way. Both ways are shown in Listing 6-8. The function shown takes a single argument: the event object that's passed in to the event handler. This function should be used at the very end of your event handler, like so: `return stopDefault( e );`—as your handler needs to also return false (which is, itself, returned from stopDefault for you).

Listing 6-8. *A Generic Function for Preventing the Default Browser Action from Occurring*

```
function stopDefault( e ) {
    // Prevent the default browser action (W3C)
    if ( e && e.preventDefault )
        e.preventDefault();

    // A shortcut for stoping the browser action in IE
    else
        window.event.returnValue = false;

    return false;
}
```

Using the stopDefault function, you can now stop any default action presented by the browser. This allows you to script some neat interactions for the user, such as the one shown in Listing 6-9. The code makes all the links on a page load in a self-contained <iframe>, rather than opening up a whole new page. Doing this allows you to keep the user on the page, and for possibly a more interactive experience.

■ **Note** Preventing a default action works for 95% of all cases in which you will want to use it. Things start to get really tricky when you move from browser to browser, due to the fact that it's up to the browser to prevent the default action (which they don't always do correctly), especially when working with preventing actions from key presses in text areas and preventing actions inside <iframe>s; other than that, things should be pretty sane though.

Listing 6-9. *Using stopDefault() to Override Browser Functionality*

```
// Let's assume that we already have an IFrame in the page
// with an ID of 'iframe'
var iframe = document.getElementById("iframe");

// Locate all <a> elements on the page
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Bind a click handler to the <a>
    a[i].onclick = function(e) {
        // Set the IFrame's location
        iframe.src = this.href;
```

```
        // Prevent the browser from ever visiting the web site pointed to from
        // the <a> (which is the default action)
        return stopDefault( e );
    };

}
```

Overriding default events is at the absolute crux of the DOM and events, which come together to form unobtrusive DOM scripting. I'll talk more about how this works, in a functional sense, in the section "Unobtrusive DOM Scripting" later in this chapter. However, it's not all perfect; a major point of contention arrives when it comes time to actually bind your event handlers to a DOM element. There are actually three different ways of binding events, some of which are better than others, all of which are discussed in the next section.

Binding Event Listeners

How to bind event handlers to elements has been a constantly evolving quest in JavaScript. It began with browsers forcing users to write their event handler code inline, in their HTML document. Thankfully that technique has since become much less popular (which is good, considering that it goes against the data abstraction principles of unobtrusive DOM scripting).

When Netscape and Internet Explorer were actively competing with each other, they each developed two separate, but very similar, event registration models. In the end, Netscape's model was modified to become a W3C standard, and Internet Explorer's stayed the same.

Today, there remain three ways of reliably registering events. The traditional method is an offshoot of the old inline way of attaching event handlers, but it's reliable and works consistently. The other methods are the IE and W3C ways of registering events. Finally, I present a reliable set of methods that developers can use to register and remove events and no longer worry about what browser is lying underneath.

Traditional Binding

The traditional way of binding events is the one that I've been using up until now in this chapter. It is by far the simplest, most compatible way of binding event handlers. To use this particular method, you attach a function as a property to the DOM element that you wish to watch. Some samples of attaching events using the traditional method are shown in Listing 6-10.

Listing 6-10. *Attaching Events Using the Traditional Method of Event Binding*

```
// Find the first <form> element and attach a 'submit' event handler to it
document.getElementsByTagName("form")[0].onsubmit = function(e){
    // Stop all form submission attempts
    return stopDefault( e );
};
```

```
// Attach a keypress event handler to the <body> element of the document
document.body.onkeypress = myKeyPressHandler;

// Attach an load event hanlder to the page
window.onload = function(){ … };
```

This particular technique has a number of advantages and disadvantages, which you must be aware of when using them.

Advantages of Traditional Binding

The following are the advantages of using the traditional method:

- The biggest advantage of using the traditional method is that it's incredibly simple and consistent, in that you're pretty much guaranteed that it'll work the same no matter what browser you use it in.
- When handling an event, the this keyword refers to the current element, which can be very useful (as demonstrated in Listing 6-5).

Disadvantages of Traditional Binding

The disadvantages of the traditional method are as follows:

- The traditional method only works with event bubbling, not capturing and bubbling.
- It's only possible to bind one event handler to an element at a time. This has the potential to cause confusing results when working with the popular window.onload property (effectively overwriting other pieces of code that have used the same method of binding events). An example of this problem is shown in Listing 6-11, where an event handler overwrites an old event handler.
- The event object argument is only available in non-Internet Explorer browsers.

Listing 6-11. *Event Handlers Overwriting Each Other*

```
// Bind your initial load handler
window.onload = myFirstHandler;

// somewhere, in another library that you've included,
// your first handler is overwritten
// only 'mySecondHandler' is called when the page finishes loading
window.onload = mySecondHandler;
```

Knowing that it's possible to blindly override other events, you should probably opt to only use the traditional means of event binding in simple situations, where you can trust all the other code that is running alongside yours. One way to get around this troublesome mess, however, is to use the modern event binding methods provided by browsers.

DOM Binding: W3C

The W3C's method of binding event handlers to DOM elements is the only truly standardized means of doing so. With that in mind, every modern browser supports this way of attaching events except for Internet Explorer.

The code for attaching a new handler function is simple. It exists as a function of every DOM element (named addEventListener) and takes three parameters: the name of the event (e.g., click), the function that will handle the event, and a Boolean flag to enable or disable event capturing. An example of addEventListener in use is shown in Listing 6-12.

Listing 6-12. *Sample Pieces of Code That Use the W3C Way of Binding Event Handlers*

```
// Find the first <form> element and attach a 'submit' event handler to it
document.getElementsByTagName("form")[0].addEventListener('submit',function(e){
    // Stop all form submission attempts
    return stopDefault( e );
}, false);

// Attach a keypress event handler to the <body> element of the document
document.body.addEventListener('keypress', myKeyPressHandler, false);

// Attach an load event hanlder to the page
window.addEventListener('load', function(){ … }, false);
```

Advantages of W3C Binding

The advantages to the W3C event-binding method are the following:

- This method supports both the capturing and bubbling phases of event handling. The event phase is toggled by setting the last parameter of addEventListener to false (for bubbling) or true (for capturing).
- Inside of the event handler function, the this keyword refers to the current element.
- The event object is always available in the first argument of the handling function.
- You can bind as many events to an element as you wish, with none overwriting previously bound handlers.

Disadvantage of W3C Binding

The disadvantage to the W3C event-binding method is the following:

- It does not work in Internet Explorer; you must use IE's attachEvent function instead.

If Internet Explorer utilized the W3C's method of attaching event handlers, this chapter would be much shorter than it is now, as there would be virtually no need to discuss alternative methods of binding events. Until that day, however, the W3C's event-binding methods are still the most comprehensive and easy to use.

DOM Binding: IE

In a lot of ways, the Internet Explorer way of binding events appears to be very similar to the W3C's. However, when you get down to the details, it begins to differ in some very significant ways. Some examples of attaching event handlers in Internet Explorer can be found in Listing 6-13.

Listing 6-13. *Samples of Attaching Event Handlers to Elements Using the Internet Explorer Way of Binding Events*

```
// Find the first <form> element and attach a 'submit' event handler to it
document.getElementsByTagName("form")[0].attachEvent('onsubmit',function(){
    // Stop all form submission attempts
    return stopDefault();
},);

// Attach a keypress event handler to the <body> element of the document
document.body.attachEvent('onkeypress', myKeyPressHandler);

// Attach an load event hanlder to the page
window.attachEvent('onload', function(){ … });
```

Advantage of IE Binding

The advantage to Internet Explorer's event-binding method is the following:

- You can bind as many events to an element as you desire, with none overwriting previously bound handlers.

Disadvantages of IE Binding

The disadvantages to Internet Explorer's event-binding method are the following:

- Internet Explorer only supports the bubbling phase of event capturing.
- The this keyword inside of event listener functions points to the window object, not the current element (a huge drawback of IE).
- The event object is only available in the window.event parameter.
- The name of the event must be named as ontype—for example, onclick instead of just requiring click.
- It only works in Internet Explorer. You must use the W3C's addEventListener for non-IE browsers.

As far as semistandard event features go, Internet Explorer's event-binding implementation is sorely lacking. Due to its many shortcomings, workarounds will continue to have to exist to force it to behave reasonably. However, all is not lost: A standard function for adding events to the DOM does exist and it will greatly ease our pain.

addEvent and removeEvent

In a contest run by Peter-Paul Koch (of http://quirksmode.org) in late 2005, he asked the general JavaScript-coding public to develop a new pair of functions, addEvent and removeEvent, which would provide a reliable way for users to add and remove events onto a DOM element. I ended up winning that contest with a very concise piece of code that worked well enough. However, afterward, one of the judges (Dean Edwards) then came out with another version of the functions that far surpassed what I wrote. His implementation uses the traditional means of attaching event handlers, completely ignoring the modern methods. Due to this fact, his implementation is able to work in a large number of browsers, while still providing all the necessary event niceties (such as the this keyword and standard event object). Listing 6-14 shows a sample piece of code, using all of the different aspects of event handling, which makes great use of the new addEvent function, including the prevention of the default browser event, the inclusion of the correct event object, and the inclusion of the correct this keyword.

Listing 6-14. *A Sample Piece of Code Using the addEvent Function*

```
// Wait for the page to finish loading
addEvent( window, "load", function(){

    // Watch for any keypresses done by the user
    addEvent( document.body, "keypress", function(e){
        // If the user hits the Spacebar + Ctrl key
        if ( e.keyCode == 32 && e.ctrlKey ) {

            // Display our special form
            this.getElementsByTagName("form")[0].style.display = 'block';

            // Make sure that nothing strange happens
            e.preventDefault();

        }
    });

});
```

The addEvent function provides an incredibly simple but powerful way of working with DOM events. Just looking at the advantages and disadvantages, it becomes quite clear that this function can serve as a consistent and reliable way to deal with events. The full source code to it can be found in Listing 6-15, which works in all browsers, doesn't leak any memory, handles the this keyword and the event object, and normalizes common event object functions.

Listing 6-15. *The addEvent/removeEvent Library Written by Dean Edwards*

```
// addEvent/removeEvent written by Dean Edwards, 2005
// with input from Tino Zijdel
// http://dean.edwards.name/weblog/2005/10/add-event/

function addEvent(element, type, handler) {
    // assign each event handler a unique ID
    if (!handler.$$guid) handler.$$guid = addEvent.guid++;

    // create a hash table of event types for the element
    if (!element.events) element.events = {};

    // create a hash table of event handlers for each element/event pair
    var handlers = element.events[type];
    if (!handlers) {
        handlers = element.events[type] = {};

        // store the existing event handler (if there is one)
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }

    }

    // store the event handler in the hash table
    handlers[handler.$$guid] = handler;

    // assign a global event handler to do all the work
    element["on" + type] = handleEvent;
};

// a counter used to create unique IDs
addEvent.guid = 1;

function removeEvent(element, type, handler) {
    // delete the event handler from the hash table
    if (element.events && element.events[type]) {
        delete element.events[type][handler.$$guid];
    }
};

function handleEvent(event) {
    var returnValue = true;

    // grab the event object (IE uses a global event object)
    event = event || fixEvent(window.event);
```

```
    // get a reference to the hash table of event handlers
    var handlers = this.events[event.type];

    // execute each event handler
    for (var i in handlers) {
        this.$$handleEvent = handlers[i];
        if (this.$$handleEvent(event) === false) {
            returnValue = false;
        }
    }

    return returnValue;
};

// Add some "missing" methods to IE's event object
function fixEvent(event) {
    // add W3C standard event methods
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};

fixEvent.preventDefault = function() {
    this.returnValue = false;
};

fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};
```

Advantages of addEvent

The advantages of Dean Edwards's addEvent event-binding method are the following:

- It works in all browsers, even older unsupported browsers.
- The this keyword is available in all bound functions, pointing to the current element.
- All browser-specific functions for preventing the default browser action and for stopping event bubbling are neutralized.
- The event object is always passed in as the first argument, regardless of the browser type.

Disadvantage of addEvent

The disadvantage of Dean Edwards's addEvent event-binding method is the following:

- It only works during the bubbling phase (since it uses the traditional method of event binding under the hood).

Considering just how powerful the addEvent/removeEvent functions are, there is absolutely no reason not to use them in your code. On top of what's shown in Dean's default code, it's really trivial to add things such as better event object normalization, event triggering, and bulk event removal, all things that are very difficult to do with the normal event structure.

Types of Events

Common JavaScript events can be classified into a couple different categories. Probably the most commonly used category is that of mouse interaction, followed closely by keyboard and form events. The following list provides a broad overview of the different classes of events that exist and can be handled in a web application. For a lot of examples of the events in action, please refer to Appendix B.

Mouse events: These fall into two categories: events that track where the mouse is currently located (mouseover, mouseout), and events that track where the mouse is clicking (mouseup, mousedown, click).

Keyboard events: These are responsible for tracking when keyboard keys are pressed and within what context—for example, tracking keyboard presses inside of form elements as opposed to key presses that occur within the entire page. As with the mouse, three event types are used to track the keyboard: keyup, keydown, and keypress.

UI events: These are used to track when users are utilizing one aspect of the page over another. With this you can reliably know when a user has begun input into a form element, for example. The two events used to track this are focus and blur (for when an object losses focus).

Form events: These relate directly to interactions that only occur with forms and form input elements. The submit event is used to track when a form is submitted; the change event watches for user input into an element; and the select event fires when a <select> element has been updated.

Loading and error events: The final class of events are those that relate to the *page* itself, observing its load state. They are tied to when the user first loads the page (the load event) and when the user finally leaves the page (the unload and beforeunload events). Additionally, JavaScript errors are tracked using the error event, giving you the ability to handle errors individually.

With these general classes of events in mind, I recommend that you actively look over the material in Appendix B where I dissect all the popular events, how they work, and how they behave in different browsers, and describe all the intricacies needed to make them do what you want.

Unobtrusive DOM Scripting

Everything that you've learned up to this point comes to one incredibly important goal: writing your JavaScript so that it interacts with your users unobtrusively and naturally. The driving force behind this style of scripting is that you can now focus your energy on writing good code that will work in modern browsers while failing gracefully for older (unsupported) browsers.

To achieve this, you could combine three techniques that you've learned to make an application unobtrusively scripted:

1. All functionality in your application should be verified. For example, if you wish to access the HTML DOM you need to verify that it exists and has all the functions that you need to use it (e.g., `if ( document && document.getElementById )`). This technique is discussed in Chapter 2.

2. Use the DOM to quickly and uniformly access elements in your document. Since you already know that the browser supports DOM functions, you can feel free to write your code simply and without hacks or kludges.

3. Finally, you dynamically bind all events to the document using the DOM and your addEvent function. Nowhere must you have something such as this: `<a href="#" onclick="doStuff();">…</a>`. This is very bad in the eyes of coding unobtrusively, as that code will effectively do nothing if JavaScript is turned off or if the user has an old version of a browser that you don't support. Since you're just pointing the user to a nonsensical URL, it will give no interaction to users who are unable to support your scripting functionality.

If it isn't apparent already, you need to pretend that the user does not have JavaScript installed at all, or that his browser may be inferior in some way. Go ahead, open your browser, visit your favorite web page, and turn off JavaScript; does it still work? How about all CSS; can you still navigate to where you need to go? Finally, is it possible to use your site without a mouse? All of these should be part of the ultimate goal for your web site. Thankfully, since you've built up an excellent understanding of how to code really efficient JavaScript code, the cost of this transition is negligible and can be done with minimal effort.

Anticipating JavaScript Being Disabled

The first goal that you should achieve is the complete removal of all inline event binding inside your HTML documents. There are a couple problem areas that you can look for in your document that frequently arise:

- If you disable JavaScript on your page and click any/all links, do they take you to a web page? Frequently developers will have URLs such as `href=""` or `href="#"`, meaning that they're working some additional JavaScript voodoo to get the users their results.

- If you disable JavaScript, do all of your forms work and submit properly? A common problem occurs when using <select>s as dynamic menus (that only work with JavaScript enabled).

Using these important lessons, you now have a web page that is completely usable for people who have JavaScript disabled and who continue to use unsupported browsers.

Making Sure Links Don't Rely on JavaScript

Now that the user can perform all the actions on the page, you need to make sure that the user is provided with adequate notice before any action is performed. When Google released Google Accelerator, which goes through all the links of a page and caches them for you, users

found that their e-mail, posts, and messages were magically being deleted for no apparent reason. This was due to the fact that developers were putting links in their pages to delete a message (for example), and then popping up a confirmation box (using JavaScript) to confirm the deletion. But Google Accelerator completely ignored that pop-up, as it should, and traversed the link anyway.

This scenario is an elaborate way of pointing you toward the HTTP specification, which is used to transport all documents and files over the Web. Most simply, a GET request occurs when you click a link; a POST occurs when you submit a form. In the specification it is stated that no GET request should have damaging side effects (such as deleting a message), which is why the Google Accelerator did what it did. It wasn't due to bad programming on Google's part, but on the part of the web application developers who created the links in the first place.

In a nutshell, all links on your site must be nondestructive. If by clicking a link you are able to delete, edit, or modify any user-owned data, you should probably be using a form to achieve that goal instead.

Watching for When CSS Is Disabled

One particularly sticky situation is the intersection between old and new browsers: browsers that are too old to support modern JavaScript techniques but are new enough to support CSS styling. A popular DHTML technique is to have an element start off as *hidden* (either with display set to none, or visibility set to hidden) and then have it fade in (using JavaScript) when the user first visits the page. However, if the user does not have JavaScript enabled, he will never see that element. A solution to this problem is shown in Listing 6-16.

Listing 6-16. *Providing a Fade-in-on-Load Technique Without Failing if JavaScript Is Disabled*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />

    <!--The instant the script is run, a new class is attached to the <html> element
            giving us the ability to know if JavaScript is enabled, or not.-->
    <script>document.documentElement.className = "js";</script>

    <!--If JavaScript is enabled, hide the block of text,
         which we will fade in later.-->
    <style>.js #fadein { display: none }</style>
</head>
<body>
    <div id="fadein">Block of stuff to fade in…</div>
</body>
</html>
```

This technique goes way beyond simple fade-in DHTML, however. The ability to know whether JavaScript is disabled/enabled and to apply styles is a huge win for careful web developers.

Event Accessibility

The final piece to take into consideration when developing a purely unobtrusive web application is to make sure that your events will work even without the use of a mouse. By doing this, you help two groups of people: those in need of accessibility assistance (vision-impaired users), and people who don't like to use a mouse. (Sit down one day, disconnect your mouse from your computer, and learn how to navigate the Web using only a mouse. It's a real eye-opening experience.)

To make your JavaScript events more accessible, anytime you use the click, mouseover, and mouseout events, you need to strongly consider providing alternative nonmouse bindings. Thankfully there are easy ways to quickly remedy this situation:

Click event: One smart move on the part of browser developers was to make the click event work whenever the Enter key is pressed. This completely removes the need to provide an alternative to this event. One point to note, however, is that some developers like to bind click handlers to submit buttons in forms to watch for when a user submits a web page. Instead of using that event, the developer should bind to the submit event on the form object, a smart alternative that works reliably.

Mouseover event: When navigating a web page using a keyboard, you're actually changing the focus to different elements. By attaching event handlers to both the mouseover and focus events you can make sure that you'll have a comparable solution for both keyboard and mouse users.

Mouseout event: Like the focus event for the mouseover event, the blur event occurs whenever the user's focus moves away from an element. You can then use the blur event as a way to simulate the mouseout event with the keyboard.

Now that you know which event pairs behave the way you want them to, you can revisit Listing 6-3 to build a hoverlike effect that works, even without a mouse, as shown in Listing 6-17.

Listing 6-17. *Attaching Pairs of Events to Elements to Allow for Accessible Web Page Use*

```
// Find all the <a> elements, to attach the event handlers to them
var li = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Attach a mouseover and focus event handler to the <a> element,
    // which changes the <a>s background to blue when the user either
    // mouses over the link, or focuses on it (using the keyboard)
    a[i].onmouseover = a[i].onfocus = function() {
        this.style.backgroundColor = 'blue';
    };
```

```
    // Attach a mouseout and blur event handler to the <a> element
    // which changes the <li>s background back to its default white
    // when the user moves away from the link
    a[i].onmouseout = a[i].onblur = function() {
        this.style.backgroundColor = 'white';
    };

}
```

In reality, adding the ability to handle keyboard events, in addition to typical mouse events, is completely trivial. If nothing else, this can help to serve as a way to help keyboard-dependant users better use your site, which is a huge win for everyone.

Summary

Now that you know how to traverse the DOM, and bind event handlers to DOM elements, and you know about the benefits of writing your JavaScript code unobtrusively, you can begin to tackle some larger applications and cooler effects.

In this chapter I started with an introduction to how events work in JavaScript and compared them to event models in other languages. Then you saw what information the event model provides and how you can best control it. We then explored binding events to DOM elements, and the different types of events that are available. I concluded by showing how to integrate some effective unobtrusive scripting techniques into any web page.

Next you're going to look at how to perform a number of dynamic effects and interactions, which make great use of the techniques that you just learned.