Advanced Event Handling

CHAPTER 39

In This Chapter

The difference between the event and Event objects

Capturing, processing, and redirecting events in Navigator

Working with Navigator and Internet Explorer event models

Once an HTML page loads, virtually nothing happens in the page without events. System and user actions make things happen, especially if those events trigger JavaScript functions. Navigator 4 extends the event mechanism that has been in scriptable browsers since the beginning, providing not only more events, but also a more sophisticated way of trapping and responding to events. This chapter focuses on the details of this new mechanism to help you understand how and when to use it in your pages.

A good deal of the imperative for implementing a deeper event mechanism came from Dynamic HTML. The possibility of hiding and showing any number of positionable elements on the screen—each of which has its own complement of event-driven document elements—is a great advantage to managing event handling across the application on a more global scale. The new event model is very good at helping with this.

The "Other" Event Object

In Chapter 33, you saw the basics of the event object. That was the event object with a lowercase "e," which is generated each time an event fires in response to some action. But an Event object with a capital "E" also exists. This object behaves like the Math object, which is always around and has some handy properties and methods ready for our scripts to use at any time. The Event object provides a series of properties (and no methods) that event handling routines use as constants.

The Event object's properties are divided into two groups. One group consists of four values representing modifier keys on the keyboard (Alt, Ctrl, Shift, Meta). The Meta key is the new Windows key on Windows computers; it's the Command (C) key on the Mac. You may recall seeing these constants used to determine whether a mouse event was fired with one or more of those keys pressed at the same time, as in the following:

```
function doEvent(evt) {
        if (evt.modifiers & Event.ALT_MASK) {
            statements for Alt key handling
        }
}
```

Names for the properties are all uppercase and must be retrieved in a reference with the Event object (`Event.ALT_MASK`, `Event.CONTROL_MASK`, `Event.SHIFT_MASK`, and `Event.META_MASK`). The other group of properties contains a large number of constants that represent event types. There exists one property for each event type used by objects in the document object model (and some events that have not yet been defined for objects, but likely will be in the future). For example, the `Event.CLICK` property is the way a script refers to a generic click event for some event-related methods.

Actual values for these properties are integer values. The integer values are of little use to your scripts, but you are free to substitute the integer values for the constant properties if you like. I won't bother listing the values here, but you can use a `for...in` loop construction with the Event object to obtain a list for yourself.

Capturing Events

The common way of assigning an event handler to an object is to use an event handler attribute in the HTML tag of the intended target of the event. But in Navigator 4, the mouse or keyboard event you assign to be captured by that attribute does not go directly to that object. Instead, the event passes through objects higher up the document object hierarchy. For example, if you define an `onClick=` event handler for a form button and the user clicks on that button, the click event first passes through the window object and the document object before reaching the button. If the button was in a form contained by a document inside a layer, the event passes through the window, main window document, layer object, and the layer's document before finally reaching the button. All of this event traveling occurs in less than a blink of an eye, so events don't seem any slower to react in Navigator 4 than they do in earlier scriptable browsers.

A quick check of the object listings for windows, documents, and layers in Appendix B reveals that these objects don't include event handlers for most of the user interactivity events that come from the user working the mouse or keyboard. But you can assign such an event handler to one of these "higher-up" objects, provided you also specifically instruct one or more of these objects to intercept events on their way to their targets. Moreover, you must instruct these objects to intercept events of a particular type, rather than all events.

Enabling event capture

All three objects just mentioned — window, document, and layer — have a `captureEvents()` method in their definitions. This is the method you use to enable event capture at any of those object levels. The method requires one or more parameters, which are the event types (as supplied by the Event object constants) that the object should capture, while letting all others pass untouched.

For example, if you want the window object to capture all keyPress events, you would include the following statement in a script that executes as the page loads:

```
window.captureEvents(Event.KEYPRESS)
```

If you want the window to capture multiple event types, string the event type constants together, separated by the pipe character:

```
window.captureEvents(Event.KEYPRESS | Event.CLICK)
```

Now you must assign an action to the event at the window's level for each event type. More than likely, you will have defined functions to execute for the event. Assign a function reference to the event handler by setting the handler property of the window object:

```
window.onKeyPress = processKeyEvent
window.onClick = processClickEvent
```

Hereafter, if a user clicks on a button or types into a field inside that window, the events will be processed by their respective window-level event handler functions.

Turning off event capture

Once you have enabled event capture for a particular event type in a document, that capture remains in effect until the page unloads or you specifically disable the capture. You can turn off event capture for each event via the window, document, or layer `releaseEvents()` method. This method takes the same kind of parameters — Event object type constants — as the `captureEvents()` method.

The act of releasing an event type simply means that events go directly to their intended targets without stopping elsewhere for processing, even if an event handler for the higher-level object is still defined. And because you can release individual event types based on parameters set for the `releaseEvents()` method, other events being captured are not affected by the release of others.

To demonstrate not only the `captureEvents()` and `releaseEvents()` methods, but other event model techniques, I present a series of several versions of the same document. Each version will implement an added feature to help you experience the numerous interactions among events and event handling methods. The document merely contains a few buttons, plus some switches to enable and disable various methods being demonstrated in the section. A layer object is also thrown into the mixture because a lot of impetus for capturing and modifying event handling comes from application of layers in a document.

Listing 39-1 is the first example, which shows the basic event capture and release from the outermost document level. A checkbox lets you enable or disable the document-level capture of click events (all checkboxes in these examples use `onMouseUp=` event handlers to avoid getting in the way of tracing click events). Because all click events are being captured by the outermost document, even clicks to the layer's buttons get trapped by the outermost document when `captureEvents()` is set.

Listing 39-1: **Event Capture and Release**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function setDocCapture(enable) {
        if (!enable) {
           document.captureEvents(Event.CLICK)
        } else {
           document.releaseEvents(Event.CLICK)
        }
}
function doMainClick(e) {
        if (e.target.type == "button") {
           alert("Captured in top document")
        }
}
document.captureEvents(Event.CLICK)
document.onclick=doMainClick
</SCRIPT>
</HEAD>
<BODY>
<B>Basic document-level capture of Event.CLICK</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" onMouseUp="setDocCapture(this.checked)"
CHECKED>Enable Document Capture
<HR>
<INPUT TYPE="button" VALUE="Button 'main1'" NAME="main1"
        onClick="alert('Event finally reached Button:' + this.name)">
</FORM>

<LAYER ID="layer1" LEFT=200 TOP=150 BGCOLOR="coral">
<HEAD>
</HEAD>
<BODY>
<FORM>
<BR><P><INPUT TYPE="button" VALUE="Button 'layerButton1'"
        NAME="layerButton1"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
<P><INPUT TYPE="button" VALUE="Button 'layerButton2'"
        NAME="layerButton2"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
</FORM>
</BODY>
</LAYER>

</BODY>
</HTML>
```

With document-level event capture turned on (the default), all click events are trapped by the document's onclick event handler property, a function that alerts the user that the event was captured by the top document. Because all click events for buttons are trapped there, even click events of the layer's buttons are trapped at the top.

In Listing 39-2, I add some code (shown in boldface) that lets the layer object capture click events whenever the outer document event capture is turned off. Inside the <LAYER> tag, a script sets the layer to capture click events. Therefore, if you disable the outer document capture, the click event goes straight to the main1 button and to the layer event capture. Event capture in the layer object prevents the events from ever reaching the buttons in the layer, unless you disable event capture for both the document and the layer.

Listing 39-2: **Document and Layer Event Capture and Release**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function setDocCapture(enable) {
        if (!enable) {
           document.captureEvents(Event.CLICK)
        } else {
           document.releaseEvents(Event.CLICK)
        }
}
function setLayerCapture(enable) {
        if (!enable) {
           document.layer1.captureEvents(Event.CLICK)
        } else {
           document.layer1.releaseEvents(Event.CLICK)
        }
}
function doMainClick(e) {
        if (e.target.type == "button") {
           alert("Captured in main.html")
        }
}
document.captureEvents(Event.CLICK)
document.onclick=doMainClick
</SCRIPT>
</HEAD>
<BODY>
<B>Document-level and/or Layer-level capture of Event.CLICK</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" onMouseUp="setDocCapture(this.checked)"
CHECKED>Enable Document Capture
<INPUT TYPE="checkbox" onMouseUp="setLayerCapture(this.checked)"
CHECKED>Enable Layer Capture
<HR>
<INPUT TYPE="button" VALUE="Button 'main1'" NAME="main1"
```

(continued)

Listing 39-2 *(continued)*

```
        onClick="alert('Event finally reached Button:' + this.name)">
</FORM>

<LAYER ID="layer1" LEFT=200 TOP=150 BGCOLOR="coral">
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function doLayerClick(e) {
        if (e.target.type == "button") {
           alert("Captured in layer1")
        }
}
layer1.captureEvents(Event.CLICK)
layer1.onclick=doLayerClick
</SCRIPT>
</HEAD>
<BODY>
<FORM>
 layer1<BR><P><INPUT TYPE="button" VALUE="Button 'layerButton1'"
        NAME="layerButton1"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
<P><INPUT TYPE="button" VALUE="Button 'layerButton2'"
        NAME="layerButton2"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
</FORM>
</BODY>
</LAYER>

</BODY>
</HTML>
```

Passing events toward their targets

If you capture a particular event type, your script may need to perform some limited processing on that event before letting it reach its intended target. For example, perhaps you want to do something special if a user clicks on an element with the Shift metakey pressed. In that case, the function that handles the event at the document level will inspect the event's `modifiers` property to determine if the Shift key was pressed at the time of the event. If the Shift key was not pressed, you want the event to continue on its way to the element that the user clicked on.

To let an event pass through the object hierarchy to its target, you use the `routeEvent()` method, passing as a parameter the event object being handled in the current function. A `routeEvent()` method does not guarantee that the event will reach its intended destination, because another object in between may have event capturing for that event type turned on and will intercept the event. That object, too, can let the event pass through with its own `routeEvent()` method.

Listing 39-3 demonstrates event routing by adding onto the document being built in previous examples. While the clickable button objects are the same, additional powers are added to the document and layer function handlers that process events that come their way. For each of these event-capturing objects, you have additional checkbox settings to allow or disallow events from passing through once they've been processed by each level.

The default settings for the checkboxes are like the ones in Listing 39-2, where event capture (for the click event) is set for both the document and layer objects. A click on any button causes the document object's event handler to process, and none other. But if you then enable the checkbox that lets the event continue, you find that click events on the layer buttons cause alerts to display from both the document and layer object event handler functions. If you then also let events continue from the layer object, a click on the button displays a third alert, showing that the event has reached the buttons. Because the `main1` button is not in the layer, none of the layer object event handling settings affect its behavior.

Listing 39-3: **Capture, Release, and Route Events**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function setDocCapture(enable) {
        if (!enable) {
           document.captureEvents(Event.CLICK)
        } else {
           document.releaseEvents(Event.CLICK)
           document.forms[0].setDocRte.checked = false
           docRoute = false
        }

}
function setLayerCapture(enable) {
        if (!enable) {
           document.layer1.captureEvents(Event.CLICK)
        } else {
           document.layer1.releaseEvents(Event.CLICK)
           document.forms[0].setLyrRte.checked = false
           layerRoute = false
        }
}
var docRoute = false
var layerRoute = false
function setDocRoute(enable) {
        docRoute = !enable
}
function setLayerRoute(enable) {
        layerRoute = !enable
}
function doMainClick(e) {
        if (e.target.type == "button") {
           alert("Captured in main.html")
```

(continued)

Listing 39-3 *(continued)*

```
            if (docRoute) {
               routeEvent(e)
            }
         }
}
document.captureEvents(Event.CLICK)
document.onclick=doMainClick
</SCRIPT>
</HEAD>
<BODY>
<B>Capture, Release, and Routing of Event.CLICK</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" NAME="setDocCap"
onMouseUp="setDocCapture(this.checked)" CHECKED>Enable Document
Capture 
<INPUT TYPE="checkbox" NAME="setDocRte"
onMouseUp="setDocRoute(this.checked)">And let event continue<P>
<INPUT TYPE="checkbox" NAME="setLyrCap"
onMouseUp="setLayerCapture(this.checked)" CHECKED>Enable Layer
Capture 
<INPUT TYPE="checkbox" NAME="setLyrRte"
onMouseUp="setLayerRoute(this.checked)">And let event continue
<HR>
<INPUT TYPE="button" VALUE="Button 'main1'" NAME="main1"
       onClick="alert('Event finally reached Button:' + this.name)">
</FORM>

<LAYER ID="layer1" LEFT=200 TOP=150 BGCOLOR="coral">
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function doLayerClick(e) {
       if (e.target.type == "button") {
          alert("Captured in layer1")
          if (layerRoute) {
             routeEvent(e)
          }
       }
}
layer1.captureEvents(Event.CLICK)
layer1.onclick=doLayerClick
</SCRIPT>
</HEAD>
<BODY>
<FORM>
 layer1<BR><P><INPUT TYPE="button" VALUE="Button 'layerButton1'"
       NAME="layerButton1"
       onClick="alert('Event finally reached Button:' +
this.name)"></P>
<P><INPUT TYPE="button" VALUE="Button 'layerButton2'"
```

```
        NAME="layerButton2"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
</FORM>
</BODY>
</LAYER>

</BODY>
</HTML>
```

In some cases, your scripts need to know if an event passed onward by `routeEvent()` method activated a function that returns a value. This is especially true if your event must return a `true` or `false` value to let an object know if it should proceed with its default behavior (for example, whether a link should activate its `HREF` attribute URL or cancel when the event handler evaluates to `return true` or `return false`). When a function is invoked by the action of a `routeEvent()` method, the return value of the destination function is passed back to the `routeEvent()` method. That value, in turn, can be returned to the object that originally captured the event.

Event traffic cop

The last scenario is one in which a higher-level object captures an event and directs the event to a particular object elsewhere in the hierarchy. For example, you could have a document-level event handler function direct every click event whose modifiers property indicates that the Alt key was pressed to a Help button object whose own `onClick=` event handler displays a help panel (perhaps shows an otherwise hidden layer).

You can redirect an event to any object via the `handleEvent()` method. This method works differently from the others described in this chapter, because the object reference of this method is the reference of the object to handle the event (with the event object being passed as a parameter like the other methods). As long as the target object has an event handler defined for that event, it will process the event as if it had received the event directly from the system (even though the event object's `target` property may be some other object entirely).

To demonstrate how this event redirection works, Listing 39-4 includes the final additions to the document being built in this chapter. It includes mechanisms that allow all click events to be sent directly to the second button in the layer (layerButton2). The previous interaction with document and layer event capture and routing is still intact, although you cannot have event routing and redirection on at the same time.

The best way to see event redirection at work is to enable both document and layer event capture (the default settings). When you click the main1 button, the event reaches only as far as the document-level capture handler. But if you then turn on the Send event to 'layerButton2' checkbox associated with the document level, a click of the main1 button reaches both the document-level event handler and layerButton2, even though the main1 button is not anywhere near the layer button in the document object hierarchy. Click other checkboxes to work with the interaction of event capturing, routing, and redirection.

Listing 39-4: **Redirecting Events**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function setDocCapture(enable) {
        if (!enable) {
           document.captureEvents(Event.CLICK)
        } else {
           document.releaseEvents(Event.CLICK)
           document.forms[0].setDocRte.checked = false
           docRoute = false
        }

}
function setLayerCapture(enable) {
        if (!enable) {
           document.layer1.captureEvents(Event.CLICK)
        } else {
           document.layer1.releaseEvents(Event.CLICK)
           document.forms[0].setLyrRte.checked = false
           layerRoute = false
        }
}
var docRoute = false
var layerRoute = false
function setDocRoute(enable) {
        docRoute = !enable
        document.forms[0].setDocShortCircuit.checked = false
        docShortCircuit = false
}
function setLayerRoute(enable) {
        layerRoute = !enable
        document.forms[0].setLyrShortCircuit.checked = false
        layerShortCircuit = false
}

var docShortCircuit = false
var layerShortCircuit = false
function setDocShortcut(enable) {
        docShortCircuit = !enable
        if (docShortCircuit) {
           document.forms[0].setDocRte.checked = false
           docRoute = false
        }
}
function setLayerShortcut(enable) {
        layerShortCircuit = !enable
        if (layerShortCircuit) {
           document.forms[0].setLyrRte.checked = false
           layerRoute = false
        }
```

```
}

function doMainClick(e) {
        if (e.target.type == "button") {
           alert("Captured in main.html")
           if (docRoute) {
              routeEvent(e)
           } else if (docShortCircuit) {

document.layer1.document.forms[0].layerButton2.handleEvent(e)
           }
        }
}
document.captureEvents(Event.CLICK)
document.onclick=doMainClick
</SCRIPT>
</HEAD>
<BODY>
<B>Redirecting Event.CLICK</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" NAME="setDocCap"
onMouseUp="setDocCapture(this.checked)" CHECKED>Enable Document
Capture 
<INPUT TYPE="checkbox" NAME="setDocRte"
onMouseUp="setDocRoute(this.checked)">And let event continue
<INPUT TYPE="checkbox" NAME="setDocShortCircuit"
onMouseUp="setDocShortcut(this.checked)">Send event to
'layerButton2'<P>
<INPUT TYPE="checkbox" NAME="setLyrCap"
onMouseUp="setLayerCapture(this.checked)" CHECKED>Enable Layer
Capture 
<INPUT TYPE="checkbox" NAME="setLyrRte"
onMouseUp="setLayerRoute(this.checked)">And let event continue
<INPUT TYPE="checkbox" NAME="setLyrShortCircuit"
onMouseUp="setLayerShortcut(this.checked)">Send event to
'layerButton2'<P>
<HR>
<INPUT TYPE="button" VALUE="Button 'main1'" NAME="main1"
       onClick="alert('Event finally reached Button:' + this.name)">
</FORM>

<LAYER ID="layer1" LEFT=200 TOP=200 BGCOLOR="coral">
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function doLayerClick(e) {
        if (e.target.type == "button") {
           alert("Captured in layer1")
           if (layerRoute) {
              routeEvent(e)
           } else if (layerShortCircuit) {
              document.forms[0].layerButton2.handleEvent(e)
```

(continued)

Listing 39-4 *(continued)*

```
            }
        }
}
layer1.captureEvents(Event.CLICK)
layer1.onclick=doLayerClick
</SCRIPT>
</HEAD>
<BODY>
<FORM>
 layer1<BR><P><INPUT TYPE="button" VALUE="Button 'layerButton1'"
        NAME="layerButton1"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
<P><INPUT TYPE="button" VALUE="Button 'layerButton2'"
        NAME="layerButton2"
        onClick="alert('Event finally reached Button:' +
this.name)"></P>
</FORM>
</BODY>
</LAYER>

</BODY>
</HTML>
```

Modifying events

A desirable capability in some scenarios would be to modify events while they are on their way to their intended targets. For example, you could capture the keyPress event for a text box, and make sure that all letters are converted to uppercase before the characters appear in the box.

Unfortunately, Netscape's event model in Navigator 4 does not support this possibility. This feature may be available in the future, but it will also probably require signed scripts. Changing an event's data could be construed as a security risk. Internet Explorer 4, however, does allow for modifying such items as a keyboard event's character without any added security.

Dueling Event Models

For their level 4 browsers, Netscape and Microsoft have gone their separate ways in expanding an event model to support requirements of flexible Dynamic HTML. While both support the standard event handler mechanism for virtually all objects in the Netscape document object model (except the Netscape-specific layer object), that is where their similarity ends.

Microsoft's document object model turns practically anything that has HTML tags around it into an object capable of supporting one or more events. In other words, you can assign an `onClick=` event handler to an `<H1>`-tagged object.

Another significant discrepancy is in the way events can ripple through the document object hierarchy. Navigator 4 has events trickling down from the window object; Internet Explorer 4 has events bubbling up from the object to the window. As a result of these differing event approaches, it is not an easy task to create a single document that meets the event programming needs of both browsers. For some applications, however, your scripts can perform a bit of browser-specific branching to achieve the same goal. I show you two examples for working with click and keyPress events.

Cross-platform modifier key check

Listing 39-5 is an enhanced version of a listing from Chapter 33's discussion of the event object's `modifiers` property. User interaction is unchanged: You can click a link, type into a text box, and click a button while holding down any combination of modifier keys. A series of four checkboxes representing the four modifier keys is at the bottom. As you click or type, the checkbox(es) of the pressed modifier key(s) become checked.

Listing 39-5: **Checking Events for Modifier Keys**

```
<HTML>
<HEAD>
<TITLE>Modifiers Event Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var isNav4, isIE4
if (parseInt(navigator.appVersion.charAt(0)) >= 4) {
       var isNav4 = (navigator.appName == "Netscape") ? true : false
       var isIE4 = (navigator.appName.indexOf("Microsoft" != -1)) ?
true : false
}

function checkMods(evt) {
       var form = document.forms[0]
       if (isNav4) {
          form.modifier[0].checked = evt.modifiers & Event.ALT_MASK
          form.modifier[1].checked = evt.modifiers & Event.CONTROL_MASK
          form.modifier[2].checked = evt.modifiers & Event.SHIFT_MASK
          form.modifier[3].checked = evt.modifiers & Event.META_MASK
       } else if (isIE4) {
          form.modifier[0].checked = window.event.altKey
          form.modifier[1].checked = window.event.ctrlKey
          form.modifier[2].checked = window.event.shiftKey
          form.modifier[3].checked = false
       }
       return false
}
</SCRIPT>
</HEAD>
<BODY>
<B>Event Modifiers</B>
<HR>
```

(continued)

Listing 39-5 *(continued)*

```
<P>Hold one or more modifier keys and click on
<A HREF="javascript:void(0)" onMouseDown="return checkMods(event)">
this link</A> to see which keys you are holding.</P>
<FORM NAME="output">
Enter some text with uppercase and lowercase letters:
<INPUT TYPE="text" SIZE=40 onKeyPress="checkMods(event)"><P>
<INPUT TYPE="button" VALUE="Click Here" onClick="checkMods(event)"><P>
<INPUT TYPE="checkbox" NAME="modifier">Alt
<INPUT TYPE="checkbox" NAME="modifier">Control
<INPUT TYPE="checkbox" NAME="modifier">Shift
<INPUT TYPE="checkbox" NAME="modifier">Meta
</FORM>
</BODY>
</HTML>
```

The concern in this chapter's listing is that Internet Explorer 4 handles the modifier notification and event object differently from Navigator 4. To start the script, I define two variables to act as flags for Navigator 4 and Internet Explorer 4. If neither browser is running the script, both flags remain null.

Since all event handlers call the same `checkMods()` function, branching is needed only in this function. For Navigator 4, the event object is passed as a parameter (`evt`) whose `modifiers` property is Bitwise ANDed with an Event object constant for each modifier key. For Internet Explorer 4, the script checks the `window.event` object property for each of three modifiers (Internet Explorer 4 does not have a metakey property). The `window.event` object is automatically set when the event occurs, so the script can simply query its properties as needed.

Cross-platform key capture

To demonstrate keyboard events in both browsers, Listing 39-6 captures the key character being typed into a text box, as well as the mouse button used to click a link or button. As with the `modifiers` property example in Listing 39-5, Navigator 4 and Internet Explorer 4 have quite different property references to reach these values. In fact, whereas Netscape combines the features of key character code and mouse button into one event object property (depending upon the event type), Internet Explorer 4 has entirely separate properties for these values.

Listing 39-6: **Checking Events for Key and Mouse Button Pressed**

```
<HTML>
<HEAD>
<TITLE>Event.which Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var isNav4, isIE4
if (parseInt(navigator.appVersion.charAt(0)) >= 4) {
```

```
        var isNav4 = (navigator.appName == "Netscape") ? true : false
        var isIE4 = (navigator.appName.indexOf("Microsoft" != -1)) ?
true : false
}
function checkWhich(evt) {
        var theKey
        if (isNav4) {
           theKey = evt.which
        } else if (isIE4) {
           if (window.event.srcElement.type == "textarea") {
              status = window.event.keyCode
           } else if (window.event.srcElement.type == "button") {
              theKey = window.event.button
           }
        }
        status = theKey
        return false
}
</SCRIPT>
</HEAD>
<BODY>
<B>Event.which Properties</B> (results in the status bar)
<HR>
<FORM NAME="output">
<P>Click on
<A HREF="javascript:void(0)" onMouseDown="return checkWhich(event)">
this link</A> or this
<INPUT TYPE="button" VALUE="Button" onClick="checkWhich(event)">
with either mouse button (if you have more than one).</P>
Enter some text with uppercase and lowercase letters:
<TEXTAREA COLS=40 ROWS=4 onKeyPress="checkWhich(event)"
WRAP="virtual"></TEXTAREA><P>
</FORM>
</BODY>
</HTML>
```

The listing starts as Listing 39-5 does by setting flags for the browser type. All event processing is handled in the `checkWhich()` function. Navigator 4 extracts the value of the `which` property. No special processing is needed for object or event type, since all I'm interested in here is the value of the `which` property. But for Internet Explorer 4, the `window.event` object has different properties for the typed key and mouse button. Therefore, the script examines the type of element that initiated the event (via the `window.event.srcElement` property). I fashioned the demonstration to show how you can use both branches of the function to dump the key code value into a single variable (`theKey`) and then treat the variable independent of the browser that generated its value.

Notice one other point about the event handler processing in Listing 39-6. The event handler function returns `false`. This lets the link use the returned value to cancel action on the link. But because both the button and textarea event handlers don't utilize the returned value (that is, they don't evaluate to `return true` or

`return false`), the default behavior is to carry out the event after the event handler function has done its processing.

Future events

It is conceivable that the Document Object Model standard or recommendation will establish a common denominator that will enable a single script to handle more complex event management in future versions of the two browsers. In the meantime, it takes a fair amount of thought and planning to minimize the effect of the two largely incompatible models.

✦ ✦ ✦