

The Event Object

Prior to Navigator 4, user and system actions — events — were captured predominantly by event handlers defined as attributes inside HTML tags. For instance, when a user clicked on a button, the click event triggered the `onClick=` event handler in the tag. That handler might invoke a separate function or perform some inline JavaScript script. Even so, the events themselves were rather dumb: Either an event occurred or it didn't. Where it occurred (that is, the screen coordinates of the pointer when the mouse button was clicked) and other pertinent event tidbits (for example, whether a keyboard modifier key was pressed at the same time) were not part of the equation. Until Navigator 4, that is.

While remaining fully backward compatible with the event handler mechanism of old, the Navigator 4 event model turns events into first-class objects whose properties automatically carry a lot of relevant information about the event when it occurs. These properties are fully exposed to scripts, allowing pages to respond more intelligently about what the user does with the page and its elements.

Another new aspect of Navigator 4's event model is that an event is no longer confined to only the intended object of the event. It is possible to capture events on a more global basis, perform some preprocessing, if necessary, and then dispatch the event either to its original target or some other object under your script control.

Why Events?

Graphical user interfaces are more difficult to program than the “old-fashioned” command-line interface. With a command-line or menu-driven system, users were intentionally restricted in the types of actions they could take at any given moment. The world was very modal, primarily as a convenience to programmers who led users through rigid program structures.

That all changed in a graphical user interface such as Windows, MacOS, Xwindows, and all others derived from the pioneering work of the Xerox Star system. The challenge for programmers is that a good user interface in this realm must make it possible for users to perform all kinds of actions at any given moment: roll the mouse, click a button, type a key,

33

CHAPTER



In This Chapter

Event support in different browser generations

How to retrieve information from an event

The many ways to assign event handlers



select text, choose a pull-down menu item, and so on. To accommodate this, a program (or, better yet, the operating system) must be on the lookout for any possible activity coming from all input ports, whether it be the mouse, keyboard, or network connection.

A common methodology to accomplish this at the operating system level is to look for any kind of event, whether it comes from user action or some machine-generated activity. The operating system or program then looks up how it should process each kind of event. Such events, however, must have some smarts about them so that the program knows what and where on the screen the event is.

Event handlers

Fortunately for scripters, the JavaScript language and document object model shield us from the dirty work of monitoring events. In a scriptable browser, an event is processed to some degree before it ever reaches anything scriptable. For instance, when a user clicks on a button, the browser takes care of figuring out where on the page the click occurred and what, if any, button object was under the pointer at the time of the click. If it was a button, then the event is sent to that button. That's where your `onClick=` event handler first gets wind that a user has done something with the button. The physical action has already taken place; now it's time for the script to do something in response to that action. If no event handler is defined for that event in that object, nothing happens beyond whatever the browser does internally (for example, highlighting the button object art).

In Navigator 2 and Internet Explorer 3, event handlers process events in the simplest possible manner. All event actions were scripted via event handler attributes inside regular HTML tags, such as the following:

```
<INPUT TYPE="text" NAME="age" SIZE="3" onChange="isNumber(this)">
```

When a user types into the above field and tabs or clicks outside of the field, the text object receives a change event from the browser, triggering the `onChange=` event handler. In the example just given, the event handler invokes a custom function that validates the entry in the field to make sure it's a number. Conveniently, a parameter conveying a reference to the text object is passed to the function. Parameters, including references to the object's containing form, are easily passed to functions this way.

Event properties

Navigator 3 brought an additional way to specify an event handler action for an object. In addition to the traditional event handler attribute in the HTML tag, the event handler was also a property of the object. You could assign a function reference to an object's event property outside of the tag, as follows:

```
document.forms[0].age.onChange = isNumber
```

A big difference with this methodology is that you do not pass arguments to the function, as you can with traditional event handler syntax. You can, however, use some advanced function object techniques to create a reference that passes a parameter to the function, provided you also explicitly include the event objects in the definition, as follows:

```
document.forms[0].age.onChange = new Function("event",
"document.forms[0].age")
```

Or, if the script is written in a `<SCRIPT LANGUAGE="JavaScript1.2">` tag set:

```
document.forms[0].age.onChange = function(event)
{isNumber(document.forms[0].age)}
```

No matter which way you use to assign a function to an event handler on the fly, this flexibility allows a script to modify the function invoked by an object's event handler even after the document has loaded — perhaps user selections in a form dictate that the form's `onsubmit` event handler should process a different presubmission function than the one specified in the tag. In Navigator 3, these event properties are all lowercase, while the corresponding event handler attribute is traditionally spelled with a capital letter after the “on” portion of the event handler name (for example, `onLoad`).

You must exercise some care in positioning scripts in your page to define event handlers in this fashion. The object must have already been loaded in the document before you can set any of its properties. Therefore, you cannot set a form element's event handlers in this way anywhere in the document that executes before the tag that generates the object in the current document object model. A safe place to set these properties is in a function invoked by the window's `onLoad` event handler. That way you know for sure all objects have completed their loading and rendering.

New Navigator — New Events

Many new events have been defined for objects in Navigator 4. For form elements, there now exist events for the components of a complete mouse click as well as keyboard events for text objects. Drag and drop is supported such that scripts can capture the user action of dragging URL text from outside of the browser to the current page. And events now signal when a user has moved or resized a browser window or frame. Table 33-1 shows the complete matrix of objects and events introduced with each generation of Navigator from 2 through 4.

Table 33-1
Event Handlers through the Navigator Ages

Object	Navigator 2	Navigator 3	Navigator 4
window	<code>onBlur</code>		<code>onDragDrop</code>
	<code>onFocus</code>		<code>onMove</code>
	<code>onLoad</code>		<code>onResize</code>
	<code>onUnload</code>		

(continued)

Table 33-1

<i>Object</i>	<i>Navigator 2</i>	<i>Navigator 3</i>	<i>Navigator 4</i>
layer			onBlur onFocus onLoad onMouseOut onMouseOver onMouseUp
link	onClick onMouseOver	onMouseOut	ondblClick onMouseDown onMouseUp
area		onMouseOut onMouseOver	onClick
image		onAbort onError onLoad	
form	onSubmit	onReset	
text, textarea, password	onBlur onChange onFocus onSelect	onKeyDown	onKeyPress onKeyUp
all buttons	onClick		onMouseDown onMouseUp
select	onBlur onChange onFocus		
fileUpload		onBlur onFocus onSelect	

Enhanced mouse events

Most clickable objects — buttons and links in particular — respond to more events in Navigator 4 than in previous browser versions. Whereas in previous versions all you could script was a complete click of the mouse button atop the object, you can now extract the mouse down (button depression) and mouse up (button release) components of the click event. If the user presses and releases the mouse button with the pointer atop the object, the sequence of events is `mouseDown`, `mouseUp`, `click`. Perhaps the biggest potential for these events is in links surrounding images. Rather than perform the mouse rollover image swap, you can swap the image on the press and release of the mouse button, just the same way as most graphical user interface buttons respond in multimedia programs.

Links can also respond to a double-click event (although this is not implemented for the Macintosh version of Navigator 4). Again, for iconic representations that simulate the workings of the desktop, a double-click action may be appropriate for your page design.

Keyboard events

At long last, Navigator provides facilities for capturing keyboard events in text fields in forms. A complete key press consists of `keyDown` and `keyUp` events. Be aware that these events are sent to the event handler before the keyboard characters are rendered in the text box. This allows your script to completely intercept keyboard input before the data reaches the box. Therefore, you might want to inspect characters to make sure only letters or only numbers reach the field.

In Navigator 4, the only keys that generate these events are the basic typewriter character keys and the Enter key. None of the navigation or function keys is capturable at this point (although some modifier keys are, as described in the discussion later in this chapter about the `which` property of the event object).

DragDrop events

Many Navigator 3 and 4 users aren't aware that you can drag text from any document from any application onto the browser window. If that text includes a URL, Navigator loads that URL's document into the browser. In Navigator 4, that user action can be monitored to some degree. The `DragDrop` event is a window event and fires every time the user drags something to the window. However, the URL is visible to your scripts only if the scripts are signed (see Chapter 40). Such URL data is deemed to be private information that requires user permission to capture.

Window modification events

One last set of events applies to the window object (and, by extension, a frame object). Navigator fires the `Move` and `Resize` events when the user changes the location or dimension of the entire browser window or frame border. Since a window's `onLoad` event handler does not fire on a window resize, this user action may be of value to you if you want to reassess the new window or frame size before reinitializing some data or reloading the page.



Note

In addition to supplying these new events, Navigator adds significant power to all events. Whenever an event occurs (whether it's by the user or the system), Navigator creates an event object. In the next section, I take up the formal definitions of this object and its properties. In Chapter 39, I cover advanced event handling in depth, particularly how to capture and redirect events before they reach their intended targets.

While Internet Explorer 4 also defines an event object in response to a user or system event, it works very differently from the Navigator event object. You can, however, share the “old-fashioned” event handler mechanism even with the newer mouse and keyboard events.

Event Object

<i>Properties</i>	<i>Methods</i>	<i>Event Handlers</i>
data	(None)	(None)
layerX		
layerY		
modifiers		
pageX		
pageY		
screenX		
screenY		
target		
type		
which		

Syntax

Accessing event properties:

eventObject.property

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

About this object

An event object is generated automatically whenever Navigator detects any one of numerous user and browser-induced actions (for example, loading a document in the window). You don't see this object, but it is there, and you can pass the object along to functions invoked by any event handler. The function can then examine the properties of the event object and process the event accordingly.

How you pass the event object to a function depends on the syntax you use to associate a function with the event. For a traditional event handler attribute in an HTML tag, the event object is passed as a keyword parameter. For example, the following tag for a text input field shows an `onKeyPress=` event handler and the passing of both the event object and the form containing the field:

```
<INPUT TYPE="text" NAME="entry" onKeyPress="handleKey(event,
this.form)">
```

Just as the `this` keyword conveys a reference to the current object, the `event` keyword conveys a reference to the event object held in memory as a result of the user pressing a keyboard key. Do not place either keyword inside quotes.

At the receiving end, the function treats the parameter like any other:

```
function handleKey(evt, form) {
    statements
}
```

You assign a local variable to the incoming event object reference (in the example I arbitrarily chose `evt`). Use that variable to access the event object's properties. If, instead of specifying the event handler in the HTML tag, I choose to specify event handling by setting the object's `onkeypress` event property, the association is made with the following syntax:

```
document.forms[0].entry.onkeypress = handleKey
```

This syntax is available from Navigator 3 or later (and Internet Explorer 4 or later), and the event property must be all lowercase for compatibility with Navigator 3. Navigator 4 also accepts the capitalized version of the property, as in

```
document.forms[0].entry.onKeyPress = handleKey
```

As mentioned earlier, this format does not allow passing parameters along with the call to the function. But the event object is a special case: It is passed automatically to the referenced function. Therefore, the function would be defined as follows:

```
function handleKey(evt) {
    statements
}
```

Event objects stay in memory only as long as they are needed for their processing. For example, if you create a button that has event handlers defined for `onMouseDown=` and `onMouseUp=` and the user makes a quick click of the button, both event objects are maintained in memory until the functions invoked by their event handlers have finished running. For a brief moment there might be an overlap in the existence of both objects, with the `mouseUp` event object standing in

readiness until the `mouseDown=` event handler finishes its processing. JavaScript handles its own garbage collection of expended event objects.

Properties

data

Value: Array of strings **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

A `DragDrop` event contains information about the URL string being dragged to the browser window. Because it is possible to drag multiple items to a window (for example, many icons representing URLs on some operating systems), the value of the property is an array of strings, with each string containing a single URL (including `file://` URLs for computer files).

URL information like this is deemed to be private data, so it is exposed only to signed scripts when the user has granted permission to read browser data. If you want your signed script to capture this information without loading the URL into the window, the event handler must evaluate to `return false`.

Example

The page in Listing 33-1 contains little more than a textarea in which the URLs of dragged items are listed. To run this script without signing the scripts, turn on codebase principals, as directed in Chapter 40.

To experiment with this listing, load the page and drag any desktop icons that represent files, applications, or folders to the window. Select multiple items and drag them all at once. Because the `onDragDrop=` event handler evaluates to `return false`, the files are not loaded into the window. If you want merely to look at the URL and allow only some to process, you would generate an `if...else` construction to return `true` or `false` to the event handler as needed. A value of `return true` allows the normal processing of the `DragDrop` event to take place after your event handler function has completed its processing.

Listing 33-1: Obtaining URLs of a DragDrop Event's data Property

```
<HTML>
<HEAD>
<TITLE>Drag and Drop</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function handleDrag(evt) {

    netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead")
    var URLArray = evt.data
```



```

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserRead")
    if (URLArray) {
        document.forms[0].output.value = URLArray.join("\n")
    } else {
        document.forms[0].output.value = "Nothing found."
    }
    return false
}
</SCRIPT>
</HEAD>
<BODY onDragDrop="return handleDrag(event)">
<B>Drag a URL to this window.</B>
<HR>
<FORM>
URLs:<BR>
<TEXTAREA NAME="output" COLS=70 ROWS=4></TEXTAREA><BR>
<INPUT TYPE="reset">
</FORM>
</BODY>
</HTML>

```

layerX
layerY
pageX
pageY
screenX
screenY

Value: Integer **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

For many (but not all) mouse-related events, the event object contains a lot of information about the coordinates of the pointer when the event occurred. In the most complex case, a click in a layer object has three distinct pairs of horizontal and vertical (x and y) coordinate values relative to the layer, the page, and the entire screen. When no layers are specified for a document, the layer and page coordinate systems are identical. Note that these values are merely geographical in nature and do not, by themselves, contain any information about the object being clicked (information held by the `event.target` property).

These mouse coordinate properties are set only with specific events. In the case of a link object, the click and all four mouse events pack these values into the

event object. For buttons, however, only the mouse events (`mousedown` and `mouseup`) receive these coordinates.

Each of the two window events (`move` and `resize`) uses one of these property pairs to convey the results of the user action involved. For example, when the user resizes a window, the `resize` event stuffs the `event.layerX` and `event.layerY` properties with the inner width and height (that is, the content area) of the browser window (you can also use the optional `event.width` and `event.height` property names if you prefer). When the user moves the window, the `event.screenX` and `event.screenY` properties contain the screen coordinates of the top-left corner of the entire browser application window.

Example

You can see the effects of the coordinate systems and associated properties with the page in Listing 33-2. Part of the page contains a three-field readout of the layer-, page-, and screen-level properties. Two clickable objects are provided so you can see the differences between an object not in any layer and an object residing within a layer. The object not confined by a layer has its layer and page coordinates the same in the event object properties.

Additional readouts display the event object coordinates for resizing and moving a window. If you maximize the window under Windows, the Navigator browser's top-left corner is actually out of sight, four pixels up and to the left. That's why the `screenX` and `screenY` values are both -4.

Listing 33-2: X and Y Coordinate Properties

```
<HTML>
<HEAD>
<TITLE>X and Y Event Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function checkCoords(evt) {
    var form = document.forms[0]
    form.layerCoords.value = evt.layerX + "," + evt.layerY
    form.pageCoords.value = evt.pageX + "," + evt.pageY
    form.screenCoords.value = evt.screenX + "," + evt.screenY
    return false
}
function checkSize(evt) {
    document.forms[0].resizeCoords.value = evt.layerX + "," +
    evt.layerY
}
function checkLoc(evt) {
    document.forms[0].moveCoords.value = evt.screenX + "," +
    evt.screenY
}
</SCRIPT>
</HEAD>
<BODY onResize="checkSize(event)" onMove="checkLoc(event)">
<B>X and Y Event Properties</B>
<HR>
<P>Click on the button and in the layer/image to see the coordinate
values for the event object.</P>
```

```

<FORM NAME="output">
<TABLE>
<TR><TD COLSPAN=2>Mouse Event Coordinates:</TD></TR>
<TR><TD ALIGN="right">layerX, layerY:</TD><TD><INPUT TYPE="text"
NAME="layerCoords" SIZE=10></TD></TR>
<TR><TD ALIGN="right">pageX, pageY:</TD><TD><INPUT TYPE="text"
NAME="pageCoords" SIZE=10></TD></TR>
<TR><TD ALIGN="right">screenX, screenY:</TD><TD><INPUT TYPE="text"
NAME="screenCoords" SIZE=10></TD></TR>
<TR><TD ALIGN="right"><INPUT TYPE="button" VALUE="Click Here"
onMouseDown="checkCoords(event)"></TD></TR>
<TR><TD COLSPAN=2><HR></TD></TR>
<TR><TD COLSPAN=2>Window Resize Coordinates:</TD></TR>
<TR><TD ALIGN="right">layerX, layerY:</TD><TD><INPUT TYPE="text"
NAME="resizeCoords" SIZE=10></TD></TR>
<TR><TD COLSPAN=2><HR></TD></TR>
<TR><TD COLSPAN=2>Window Move Coordinates:</TD></TR>
<TR><TD ALIGN="right">screenX, screenY:</TD><TD><INPUT TYPE="text"
NAME="moveCoords" SIZE=10></TD></TR>
</TABLE>
<LAYER NAME="display" BGCOLOR="coral" TOP=100 LEFT=240 HEIGHT=250
WIDTH=330>
<A HREF="javascript:void(0)" onClick="return checkCoords(event)">
<IMG SRC="nile.gif" WIDTH=320 HEIGHT=240" BORDER=0></A>
</LAYER>
</BODY>
</HTML>

```

Related Items: Window and layer object move and resize methods.

modifiers

Value: Constant **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

The `modifiers` property of the event object refers to the modifier keys that can be pressed while clicking or typing. Modifier keys are Alt (also the Option key on the Macintosh keyboard), Ctrl, Shift, and what is known as a meta key (for example, the Command key, ⌘, on the Macintosh keyboard, and the Windows key on the PC keyboard). You can use this property to find out if one or more modifier keys were pressed at the time the event occurred.

Values for these keys are integer values designed in such a way that any combination of keys generates a unique value. Fortunately, you don't have to know anything about these values, because JavaScript supplies some plain-language constants (properties of a global Event object always available behind the scenes) that a script can apply to the property value passed with the object. The constant

names consist of the key name (all uppercase), followed by an underscore and the uppercase word `MASK`. For example, if the Alt key is pressed by itself or in concert with other modifier keys, you can use the bitwise AND operator (`&`) and the `Event.ALT_MASK` constant to test for the presence of the Alt key in the property value:

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

Modifiers are not available with every event. You can capture them with `mouseDown` and `mouseUp` events in buttons and links. The only click event offering modifiers is with the button objects. Keyboard events in text objects also include these modifiers. But be aware that accelerated keyboard combinations (for example, `Ctrl+Q/⌘-Q` for Quit) are not trappable by JavaScript event mechanisms.

Example

You see how a variety of object event handlers work with the `modifiers` property in Listing 33-3. A link, text box, and button have event handlers set up to pass the event object to a function that displays the modifier key(s) being pressed at the time of the event. The script contains examples of trapping for all four modifier keys with their constant values.

Listing 33-3: Modifiers Property

```
<HTML>
<HEAD>
<TITLE>Modifiers Event Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function checkMods(evt) {
    var form = document.forms[0]
    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
    return false
}
</SCRIPT>
</HEAD>
<BODY>
<B>Event Modifiers</B>
<HR>
<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>

```

target

Value: Object Reference **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

Every event has a property containing a reference to the object that was clicked on, typed into, or otherwise acted upon. Most commonly, this property is examined when you set up a page to trap for events at the window, document, or layer level, as described in Chapter 39. The `target` property lets you better identify the intended destination of the event while handling all processing for that type of event in one place. With a reference to the target object at hand in this property, your scripts can extract and/or set properties of the object directly.

type

Value: String **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

An event object's type is the name of the event that generated the event object. An event name is the same as the event handler's name, less the "on" prefix. Therefore, if a button's `onClick=` event handler is triggered by a user's click, then the event type is `click` (all lowercase). If you create a multipurpose function for handling events, you can extract the `event.type` property to help the function decide how to handle the current event. This sounds like a good job for the `switch` control structure.

which

Value: Integer **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✓			

The value of the `which` property depends on the event type: a mouse button indicator for mouse events and a character key code for keyboard events.

For a mouse-related event, the `event.which` property contains either a 1 for the left (primary) mouse button or a 3 for the right (secondary) mouse button. Most Macintosh computers have only a one-button mouse, so exercise care in designing pages that rely on the second mouse button. Even on Windows and other platforms, you must program an object's `onMouseDown=` event handler to return false for the secondary button to be registered instead of a browser pop-up menu appearing on screen.

Keyboard events generate the ISO-Latin character code for the key that has been pressed. This value is an integer between 0 and 255. If your script needs to look at the actual character being typed, rather than the key code, use the `String.fromCharCode()` method (see Chapter 26) to make the conversion.

Example

Listing 33-4 provides a readout in the status bar for the `event.which` property for an `onMouseDown=` event handler of a link and an `onKeyPress=` event handler of a textarea. As you click the link (with each mouse button if you have more than one), the value of the `event.which` property appears in the status bar. The same goes for displaying the key character code as you type into the textarea. Notice that carriage returns and spaces have codes. The Enter key is 13; Ctrl+Enter is 10.

Listing 33-4: Event.which Property

```
<HTML>
<HEAD>
<TITLE>Event.which Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function checkWhich(evt) {
    status = evt.which
    return false
}
</SCRIPT>
</HEAD>
<BODY>
<B>Event.which Properties</B> (results in the status bar)
<HR>
<P>Click on
<A HREF="javascript:void(0)" onMouseDown="return checkWhich(event)">
this link</A> with either mouse button (if you have more than
one).</P>
<FORM NAME="output">
Enter some text with uppercase and lowercase letters:
<TEXTAREA COLS=40 ROWS=4 onKeyPress="checkWhich(event)"></TEXTAREA><P>
</FORM>
</BODY>
</HTML>
```

