Chapter 48: Internet Explorer Behaviors

In This Chapter

Introducing IE behaviors

Understanding the structure of behavior XML files

Exploring behavior samples

Internet Explorer 5 for Windows was the first browser to deploy what Microsoft calls *behaviors*. Microsoft and others have proposed the behaviors concept to the W3C, and it could some day become one of the W3C standard recommendations. Such a standard might not be implemented exactly the way Microsoft currently implements behaviors, but most of the concepts are the same, and the syntax being discussed so far is similar. While there is no guarantee that the W3C will adopt behaviors as a standard, you will see that the concept seems to be a natural extension to the work that has already been adopted for both CSS and XML. Even though behaviors run only on Windows versions of IE5+ (as of this writing anyway), that browser family and operating system are pervasive enough to warrant an extended description of how behaviors work.

The W3C effort is called Behavioral Extensions to CSS. For the latest document describing the work of the participants of the standards discussions, visit `http://www.w3.org/TR/becss`.

Style Sheets for Scripts

You can best visualize what a behavior is in terms of the way you use style sheets. Consider a style sheet rule whose selector is a tag or a class name. The idea behind the style sheet is that one rule, which can define dozens of rendering characteristics of a chunk of HTML content, can be applied to perhaps dozens, if not hundreds, of elements within the document. A corporation may design a series of rules for the way its Web documents will look throughout the Web site. If the designer decides to alter the font family or color for, say, H1 elements, then that change is made in one place (the external style sheet file), and the impact is felt immediately across the entire site. Any page that includes an H1 element renders the header with the newly modified style.

Imagine now that instead of visual styles associated with an element, you want to define a *behavioral style* for a particular group of elements. A behavioral style is the way an element responds to predominantly user interaction with the element. For example, if the design specifications for a Web site indicate that all links should have their text colored a certain way when at rest, but on mouse rollovers, the text color changes to a more contrasting color, the font weight increases to bold, and the text becomes underlined.

Those modifications require scripts to change the style properties of the element in response to the mouse action of the user. The scripts that fire in response to specific user actions (events) are written in an external file known as a behavior, and a behavior is associated with an element, class, or tag through the same CSS syntax that you use for other style attributes.

A behavior, of course, assumes that its scripts can work with whatever HTML element is associated with the behavior. Just as it would be illogical to associate the `tableLayout` style attribute with an element that wasn't a TABLE, so, too, would it be illogical to associate a behavior, whose scripts employed TABLE object properties and methods, to a P element. Even so, a well-designed behavior can obtain details about the element being manipulated through the element object's properties. The better you are at writing generalizable JavaScript functions, the more successful you will be in implementing behaviors.

Embedding Behavior Components

IE treats each behavior as a component, or add-on building block for the browser. IE5 comes equipped with a handful of behaviors built into the browser (the so-called default behaviors, which happen to rely on specific XML elements embedded in a document). Behaviors that you create most likely exist as separate files on the server, just like external `.css` and `.js` files do. The file extension for a behavior file is `.htc` (standing for HTML Component).

Linking in a behavior component

To associate a behavior with any single element, class of elements, or tag as the page loads, use CSS rule syntax and the IE-specific `behavior` attribute. The basic syntax is as follows:

```
selector {behavior:url(componentReference)}
```

As with any style sheet rule, you can combine multiple rule attributes, delimiting them with semicolons. The format of the *`componentReference`* depends on whether you are using one of the IE default behaviors or a behavior you've written to an external file. For default behaviors, the reference is in the format:

```
#default#componentName
```

For example, if you want to associate the `download` behavior with any element of class `downloads`:

```
.downloads {behavior:url(#default#download)}
```

Relative or absolute URIs to external `.htc` files can also be specified. For example, if your site contains a directory named `behaviors` and a file named `hilite.htc`, the style sheet rule from the root directory is:

```
.hiliters {behavior:url(behaviors/hilite.htc)}
```

As with all CSS style sheet rules, behaviors can be specified in a STYLE element of the page, in the STYLE attribute of an individual element, or in a rule defined inside an imported .css file.

Enabling and disabling behaviors

In Chapter 15, you can find details of IE5/Windows methods for all HTML elements that let scripts manage the association of a behavior with an element after the page has loaded. Invoking the addBehavior() method on an element assigns an external .htc file to that element. When you no longer need that behavior associated with the element, invoke the removeBehavior() method.

Component Structure

An .htc behavior file is a text file consisting of script statements inside a <SCRIPT> tag set and some special XML tags that IE5/Windows knows how to parse. You create .htc files in the same kind of plain text editor that you use for external .js or .css files.

Script statements

Unlike external .js files, an .htc behavior file includes <SCRIPT> tags, which surround any JavaScript (or VBScript, if you like) statements that control the behavior. Because a behavior most typically is written to control one or more aspects of the HTML element to which it is connected, statements tend to operate only on the associated object element. A special reference — element — is used to refer to the element object itself (much like the way the this keyword in a custom object's method self-refers to the object associated with the method).

If your behavior will be modifying either the content or style of the element, use the element reference as a foundation to the reference to one of that element object's properties or methods. For example, if a statement in a behavior needs to set the style.visibility property so that the element hides itself, the statement in the behavior script is:

```
element.style.visibility = "hidden"
```

Any valid reference from the point of view of the element object is fair game, including references to the element's parentElement, even though the parent element is not explicitly associated with the behavior.

Variable scope

Except for the special element reference, script content of a behavior is completely self-contained. You can define global variables in the behavior that are accessible to any script statement in the behavior. But a global variable in a behavior does not become a

global variable for the main document's scripts to use. You can expose variables so that scripts outside of the behavior can get to them (as described below), but this exposure is not automatic.

Most of the script content of a behavior consists of functions that usually interact in some fashion with the associated element (via the element's properties and/or methods). Local variables in functions have the same scope and operate just like they do in regular script functions. Global variables you define in a behavior, if any, are usually there for the purpose of preserving values between separate invocations of the functions.

Assigning event handlers

Functions in a behavior are triggered from outside the behavior through two means: event handlers and direct invocation of functions declared as public (described in the next section). Event handler binding is performed in a way that is not used elsewhere in the IE4+ DOM. Each event type (for example, `onMouseOver`, `onKeyPress`) requires its own special XML tag at the top of the behavior file. The format for the event handler tag is as follows:

```
<PUBLIC:ATTACH EVENT="eventName" ONEVENT="behaviorFunctionName()" />
```

As the behavior loads, the `PUBLIC:ATTACH` tag instructs the browser to expose to the "public" (that is, the world outside of the behavior) an event type (whose name always begins with the "on" prefix in the IE4+ event model); whenever an event of that type reaches the behavior's element, then the function (defined within the behavior file) is invoked. In XML terminology, the `PUBLIC:` part of the tag is known as a *namespace*, and IE includes a built-in parser for the `PUBLIC` namespace. Notice, too, the XML syntax at the end of the tag that allows a single set of angle brackets to act as a start and end tag set (there is no content for this tag, just the attributes and their values).

To demonstrate, imagine that a behavior has a function named `underlineIt()`, which sets the `element.style.textDecoration` property to underline. To get the element to display the underline decoration as the user rolls the mouse atop the element, bind this function to the element's `onMouseOver` event handler as follows:

```
<PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="underlineIt()" />
```

If you compare the wording of the opening part of the tag, you may recognize a connection to the IE4+ event model's `attachEvent()` method of all HTML elements (Chapter 15). You can have as many event binding tags as your element needs. To invoke multiple functions in response to a single event type, simply add the subsequent function invocation statements to the `ONEVENT` attribute, separating the calls by semicolons (the same as with regular JavaScript statement delimiters).

Exposing properties and methods

XML tags with the `PUBLIC:` namespace are also used (with different attributes) to expose a behavior's global variables as properties of the element and a behavior's

functions as methods of the element. The syntax for both types of "public" announcements is as follows:

```
<PUBLIC:PROPERTY NAME="globalVarName" />
<PUBLIC:METHOD NAME="functionName" />
```

Values for both items are string versions of references to the variable and function (no parentheses). Again, you can define as many properties and methods for a behavior as you need.

As soon as a property and/or method is made public in a behavior, scripts from outside the behavior can access those items as if they were properties or methods of the element associated with the behavior:

```
document.all.elementID.behaviorProperty
document.all.elementID.behaviorMethod()
```

If you associate a behavior with a style sheet class selector, and several document elements share that class name, each one of those elements gains the public properties and methods of that behavior, accessible through references to the individual elements. That's because a behavior's scripts are written to read or modify properties of whatever element receives a bound event or is referenced along the way to the public property or method.

Behavior Examples

The two following examples are intentionally simple to help you grasp the concepts of behaviors if they are new to you. The first example interacts with multiple elements strictly through event binding; the second example exposes a property and method that the main page's scripts access to good effect.

Example 1: Element dragging behavior

This book contains several examples of how to script a page to let a user drag an element around the browser window (Chapters 31 and 56 in particular). In all those examples, the dragging code and event handling was embedded in some fashion into the page's scripts. The first example of a behavior, however, drives home the notion of separating an element's behavior from its content (just as a CSS2 style sheet separates an element's appearance from its content).

Imagine that it's your job to design a page that employs three draggable elements. Two of the elements are images, while the third is a panel layer that also includes a form. If you haven't scripted DHTML before, this may sound like a daunting task at first, one rife with the possibility of including multiple versions of the same scripts to accommodate different kinds of draggable elements.

Now imagine that to the rescue comes a scripter who has built a behavior that takes care of all of the dragging scripting for you. All you do is assign that behavior by way of one

attribute of each draggable element's style sheet rule. Absolutely no other scripting is required on the main page to achieve the element dragging.

Listing 48-1 shows the behavior file (`drag.htc`) that controls basic dragging of a positionable element on the page. You may recognize some of the code as an IE4+ version of the cross-browser dragging code used elsewhere in this book (for a blow-by-blow account of these functions, see the description of the map puzzle game in Chapter 56). The names of the three operative functions and the basic way they do their jobs are identical to the other dragging scripts. Event binding, however, follows the behavior format through the XML tags. All interaction with the outside world occurs through the "public" event handlers.

Listing 48-1
An Element Dragging Behavior

```
<PUBLIC:ATTACH EVENT="onmousedown" ONEVENT="engage()" />
<PUBLIC:ATTACH EVENT="onmousemove" ONEVENT="dragIt()" />
<PUBLIC:ATTACH EVENT="onmouseup"   ONEVENT="release()" />
<PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="setCursor()" />
<PUBLIC:ATTACH EVENT="onmouseout"  ONEVENT="release();restoreCursor()" />

<SCRIPT LANGUAGE="JScript">
// global declarations
var offsetX = 0
var offsetY = 0
var selectedObj
var oldZ, oldCursor

// initialize drag action on mousedown
function engage() {
    selectedObj = (element == event.srcElement) ? element : null
    if (selectedObj) {
        offsetX = event.offsetX - element.document.body.scrollLeft
        offsetY = event.offsetY - element.document.body.scrollTop
        oldZ = element.runtimeStyle.zIndex
        element.style.zIndex = 10000
        event.returnValue = false
    }
}

// move element on mousemove
function dragIt() {
    if (selectedObj) {
        selectedObj.style.pixelLeft = event.clientX - offsetX
        selectedObj.style.pixelTop = event.clientY - offsetY
        event.cancelBubble = true
        event.returnValue = false
    }
}

// restore state on mouseup
function release() {
    if (selectedObj) {
        selectedObj.style.zIndex = oldZ
    }
    selectedObj = null
}

// make cursor look draggable on mouseover
function setCursor() {
    oldCursor = element.runtimeStyle.cursor
```

```
   element.style.cursor = "hand"
}

// restore cursor on mouseout
function restoreCursor() {
    element.style.cursor = oldCursor
}
</SCRIPT>
```

Notice a subtlety in Listing 48-1 that is implied by the element-specific scope of a behavior. Two statements in the `engage()` function need to reference scroll-related properties of the `document.body` object. Because the only connection between the behavior and the document is via the `element` reference, that reference is used along with the `document` property (a property of every HTML element object in IE4+, as shown in Chapter 15). From there, the `body` object and the required properties can be accessed.

Listing 48-2 is a simple page that contains three elements that are associated with the `drag.htc` behavior through a style sheet rule definition (for the `draggable` class). The document is incredibly uncomplicated. Even the `drag.htc` file isn't very big. But together they produce a far more interesting page for the user than a couple of static images and a form.

Listing 48-2
Three Draggable Elements Using the Behavior

```
<HTML>
<HEAD>
<STYLE TYPE="text/css">
.draggable {position:absolute; behavior:url(drag.htc)}
#img1 {left:150px; top:150px}
#img2 {left:170px; top:170px}
#txt1 {left:190px; top:190px; background-color:aqua; width:150px; height:50px;
text-align:center}
</STYLE>
</HEAD>

<BODY>
<H1>IE5+ Behavior Demo (Dragging)</H1>
<HR>
<IMG CLASS="draggable" ID="img1" SRC="cpu1.gif">
<IMG CLASS="draggable" ID="img2" SRC="desk3.gif">
<DIV CLASS="draggable" ID="txt1">A form inside a DIV element.
    <FORM>
        <INPUT TYPE="button" VALUE="Does Nothing">
    </FORM>
</DIV>
</BODY>
</HTML>
```

Obviously, the dragging example here is very rudimentary. It isn't clear from the sample code what the user gets from the page, other than the joy of moving things around. If you were designing an application that genuinely benefits from draggable objects (for example, the map puzzle in Chapter 56), you can easily enhance the behavior to perform actions, such as snapping a dragged element into place when it is within a few pixels of its proper destination. For such an implementation, the behavior can be given some extra global variables, akin to the values assigned to the state objects in Chapter 56, including

the pixel coordinates of the ideal destination for a dragged element. An `onLoad` event handler for the page can fire a public `init()` function in each element's behavior to assign those coordinate values. Any event that can bubble (such as mouse events) does so from the behavior to the target. Therefore, you can extend the event action of the behavior by adding a handler for the same event to the element outside of the behavior.

Example 2: Text rollover behavior

In the second example, you see how a behavior exposes a global variable and function as a public property and method, respectively. The demonstration reinforces the notion that even if a single behavior file is associated with multiple elements (for example, the elements share the same class, and the behavior is assigned to the class), each behavior maintains its own variable values, independent of the other elements and their behaviors.

The nature of this behavior is to set the `color` style property of the associated element to either a default color (red) or to another color that has been passed into the behavior via one of its public methods. The color setting is preserved in one of the behavior's global variables, and that variable is exposed as a public property.

Listing 48-3 shows the `.htc` behavior file's content. Only two events are bound to this behavior: `onmouseover` and `onmouseout` — the typical rollover events. The `onMouseOver` event invokes the `makeHot()` function, while the `onMouseOut` event invokes the `makeNormal()` function. Before the `makeHot()` function makes any changes to the `color` and `fontWeight` style properties of the element, existing settings are preserved in (non-public) global variables in the behavior. This allows the `makeNormal()` function to restore the original settings, regardless of what document styles may be applied to the element in a variety of pages. That's something to keep in mind when you design behaviors: they can be deployed in pages controlled by any number of style sheets. Don't assume any basic style setting; instead, use the `currentStyle` property to read and preserve the effective property values before touching them with your behavior's modification scripts.

Neither of the event handler functions are exposed as public methods. This was a conscious decision for a couple of reasons. The most important reason is that both functions rely on being triggered by a known event occurring on the element. If either function were invoked externally, the event object would contain none of the desired information. Another reason behind this is from a common programming style for components that protects inner workings, while exposing only those methods and properties that are "safe" for others to invoke. For this code, the public method does little more than set a property. It's an important property, to be sure, and one of the protected functions relies on it. But by allowing the public method little room to do any damage to other execution of the behavior, the design makes the behavior component that more robust.

Assigning a color value to the public property and passing one as a parameter to the public method accomplishes the same result in this code. As you will see, the property

gets used in the demonstration page to retrieve the current value of the global variable. In a production behavior component, the programmer would probably choose to expose this value strictly as a read/write property or expose two methods, one for getting and one for setting the value. The choice would be at the whim of the programmer's style and would likely not be both. Using a method, however, especially for setting a value, creates a framework in which the programmer can also perform validation of the incoming value before assigning it to the global variable (something the example here does not do).

Listing 48-3
Rollover Behavior (makeHot.htc)

```
<PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="makeHot()" />
<PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="makeNormal()" />
<PUBLIC:PROPERTY NAME="hotColor" />
<PUBLIC:METHOD NAME="setHotColor" />
<SCRIPT LANGUAGE="JScript">
var oldColor, oldWeight
var hotColor = "red"

function setHotColor(color) {
    hotColor = color
}

function makeHot() {
    if (event.srcElement == element) {
        oldColor = element.currentStyle.color
        oldWeight = element.currentStyle.fontWeight
        element.style.color = hotColor
        element.style.fontWeight = "bold"
    }
}

function makeNormal() {
    if (event.srcElement == element) {
        element.style.color = oldColor
        element.style.fontWeight = oldWeight
    }
}
</SCRIPT>
```

To put the public information and the behavior, itself, to work, a demonstration page includes three spans within a paragraph that are associated with the behavior. Listing 48-4 shows the code for the demo page.

In addition to the text with rollover spans, the page contains two SELECT controls, which let you assign a separate color to each of the three elements associated with the behavior. The first SELECT element lets you choose one of the three elements. Making that choice invokes the `readColor()` function in the same page. This is the function that reads the `hotColor` public property of the chosen span. That color value is used to select the color name for display in the second SELECT element. If you make a choice in the list of colors, the `applyVals()` function invokes the public `setHotColor()` method of the element currently selected from the list of elements. Rolling the mouse over that element now highlights in the newly selected color, while the other elements maintain their current settings.

Listing 48-4

Applying the Rollover Behavior

```
<HTML>
<HEAD>
<STYLE TYPE="text/css">
.hotStuff {font-weight:bold; behavior:url(makeHot.htc)}
</STYLE>
<SCRIPT LANGUAGE="JavaScript">
function readColor(choice) {
    var currColor = document.all(choice.value).hotColor
    var colorList = choice.form.color
    for (var i = 0; i < colorList.options.length; i++) {
        if (colorList.options[i].value == currColor) {
            colorList.selectedIndex = i
            break
        }
    }
}
function applyVals(form) {
    var elem = form.elem.value
    document.all(elem).setHotColor(form.color.value)
}
</SCRIPT>
</HEAD>

<BODY>
<H1>IE5+ Behavior Demo (Styles)</H1>
<HR>
<FORM>
Choose Hilited Element:
<SELECT NAME="elem" onChange="readColor(this)">
    <OPTION VALUE="elem1">First
    <OPTION VALUE="elem2">Second
    <OPTION VALUE="elem3">Third
</SELECT>
Choose Hilite Color:
<SELECT NAME="color" onChange="applyVals(this.form)">
    <OPTION VALUE="red" SELECTED>Red
    <OPTION VALUE="blue">Blue
    <OPTION VALUE="green">Green
</SELECT>
</FORM>
<P>Lorem ipsum dolor sit amet, <SPAN ID="elem1"
CLASS="hotStuff">consectetaur</SPAN> adipisicing elit, sed do eiusmod tempor
incididunt ut <SPAN ID="elem2" CLASS="hotStuff">labore et dolore magna
aliqua</SPAN>. Ut enim adminim veniam, quis nostrud exercitation ullamco laboris
<SPAN ID="elem3" CLASS="hotStuff">nisi ut aliquip ex ea commodo
consequat</SPAN>.</P>
</DIV>
</BODY>
</HTML>
```

Behaviors are not the solution for every scripting requirement. As demonstrated here, they work very well for generic style manipulation, but you are certainly not limited to that sphere. By having a reference back to the element associated with the behavior, and then to the document that contains the element, a behavior's scripts can have free run over the page — provided the actions are either generic among any page or generic among a design template that is used to build an entire Web site or application.

Even if you don't elect to use behaviors now (perhaps because you must support browsers other than IE/Windows), they may be in your future. Behaviors are fun to think about and also instill good programming practice in the art of creating reusable, generalizable code.

For More Information

In addition to the address of W3C activity on behaviors, Microsoft devotes many pages of its developer site to behaviors. Here are some useful pointers.

Overview:

`http://msdn.microsoft.com/workshop/author/behaviors/overview.asp`

Using DHTML Behaviors:

`http://msdn.microsoft.com/workshop/author/behaviors/howto/using.asp`

Default Behaviors Reference:

`http://msdn.microsoft.com/workshop/author/behaviors/reference/reference.asp`

IE5.5 Element Behaviors (an extension to the original behaviors):

`http://msdn.microsoft.com/workshop/author/behaviors/overview/elementb_ovw.asp`

Each of these locations ends with yet more links to related pages at the Microsoft Developer Network (MSDN) Web site.