CHAPTER 42

Netscape Dynamic HTML and JavaScript Extensions

In This Chapter

JavaScript style sheets

Dynamic positioning

The map game tailored for Navigator 4

While Netscape Navigator 4 provides support for the Cascading Style Sheets, Level 1 recommendation by W3C, the company created its own implementation of style sheets, sometimes called JavaScript Style Sheets, or JSS's. It's not that these style sheets become part of the scripts in your document, but rather they adopt many of the syntactical traditions of JavaScript. This is in contrast to the new syntax (characterized by the property-value pair) of CSS1. In the arena of dynamically positionable objects, Navigator 4 adheres in part to the CSS-P recommendation and also offers a new HTML tag, the `<LAYER>` tag, to create such objects.

This chapter introduces the JavaScript language extensions employed by JavaScript Style Sheets. Because this is not the same kind of scripting that you've been reading about in this book, the coverage is intended merely to acquaint you with the syntax in case you are not aware of it. More coverage is accorded the positioning aspects of the `<LAYER>` tag because these items are scriptable from your regular scripts. At the end of the chapter, I provide a Navigator-specific version of the map puzzle game demonstrated in cross-platform format in Chapter 41.

JavaScript Styles

As with Cascading Style Sheets, JavaScript Style Sheets define layout characteristics for the content of an HTML tag. A style defines how the browser renders the page as it loads. Any adjustment to the content after the page loads requires scripting beyond the scope of style sheets.

If you are familiar with Cascading Style Sheets (as defined in the published W3C specification available at `http://www.w3.org/pub/WWW/TR/REC-CSS1`), you know

that a typical way for a style sheet to be defined is with the `<STYLE>` tag whose `TYPE` attribute is "text/css", as follows:

```
<STYLE TYPE="text/css">
       H1 {color:red}
</STYLE>
```

Rules inside the style tag specify the tag type and declarations of properties and values that override default browser rendering settings.

Navigator provides an alternate syntax that follows the dot syntax model of JavaScript. Its style type is "text/javascript", as follows:

```
<STYLE TYPE="text/javascript">
       tags.H1.color="red"
</STYLE>
```

Within the context of the `<STYLE>` tag, Navigator recognizes a tags object, which can contain any known tag name and one of that tag's properties. Internet Explorer 4 does not recognize the "text/javascript" style type, nor the JavaScript-like syntax within a `<STYLE>` tag set. Moreover, while Netscape's JavaScript objects used in `<STYLE>` tags are exposed to scripts written inside `<SCRIPT>` tag sets, their values cannot be modified to change the rendering of the object on the fly.

Netscape has defined several objects for style definitions. Many of them assist in creating groups of styles and exceptions to the rules defined by other styles.

The tags object

Every document object has a tags object. But since the tags object is always at the document level, the `document` reference is not needed inside the `<STYLE>` tag (although it is required if you intend to read a style property from inside a `<SCRIPT>` tag).

The tags object represents all the tags in the current document. To assign a style to a tag, you must build a reference to that tag and its property according to the following format:

```
tags.tagName.propertyName = value
```

You can assign only one property per statement.

The `tagName` part of the statement is the text you normally find between the angle brackets of a tag, such as `P` for the `<P>` tag or `BODY` for the `<BODY>` tag. Unlike the usually case-sensitive JavaScript, these tag names can be in upper- or lowercase. Tags you can use for this purpose are only those that come in a pair of opening and closing tags. These pairs delimit the range of content affected by a particular style. You can, however, apply some of your scripting knowledge to JavaScript inside the `<STYLE>` tag. For example, if you have many properties you wish to set for a single tag, the long way to do it would be

```
<STYLE TYPE="text/javascript">
       tags.p.color = "green"
       tags.p.borderColor = "blue"
       tags.p.borderStyle = "3D"
       tags.p.borderWidths("2px","2px","2px","2px")
</STYLE>
```

However, you can use the `with` operator to shorten the references:

```
<STYLE TYPE="text/javascript">
with(tags.p) {
        color = "green"
        borderColor = "blue"
        borderStyle = "3D"
        borderWidths("2px","2px","2px","2px")
}
</STYLE>
```

The classes object

Defining style properties for a given tag could be limiting if you intend to reuse that tag with a different style. For example, consider a page that contains many text paragraphs designed with a specific margin and font color. You decide you want to insert in a few places paragraphs that follow a style that increases the left and right margins and sets the text to a bold font in a different color. The cumbersome way to accomplish this would be to keep changing the style definitions throughout the document. A better way is to define different classes of the same tag. To define a class, you use the classes object in a style rule. The format for this kind of statement is as follows:

```
classes.className.tagName.propertyName = value
```

For example, to define two different classes for the `<P>` tag, you would use a construction like the following:

```
<STYLE TYPE="text/javascript">
        classes.normal.p.margins("2em","3em","2em","3em")
        classes.normal.p.textAlign = "justify"
        classes.inset.p.margins("2em","10em","2em","10em")
        classes.inset.p.textAlign = "center"
        classes.inset.p.fontWeight = "bold"
</STYLE>
```

To apply a class to a tag, use the CLASS attribute for the tag:

```
<P CLASS="normal">...</P>
<P CLASS="inset">...</P>
```

In an abstract way, the classes object lets you define new versions of tags, although the tag name stays the same. Importantly, it means you can define multiple styles for a given tag, and then deploy each style where needed throughout the document.

The ids object

One more object, the ids object, lets you assign a single style property to an identifier to make it easy to make exceptions to other styles. This works best when a tag is assigned a group of style properties, but your document needs two versions of the same style that differ by only one or two style properties. To create an ids object, use the following syntax format:

```
ids.idName.propertyName = value
```

As an example of the ids object in use, consider the two paragraph classes defined in the preceding section. If I also want an inset class that looks just like the insert class except the `fontWeight` property is `normal` and a `fontStyle` property `italic`, the entire definition would be as follows:

```
<STYLE TYPE="text/javascript">
       classes.normal.p.margins("2em","3em","2em","3em")
       classes.normal.p.textAlign = "justify"
       classes.inset.p.margins("2em","10em","2em","10em")
       classes.inset.p.textAlign = "center"
       classes.inset.p.fontWeight = "bold"
       ids.altInset.fontWeight = "normal"
       ids.altInset.fontStyle = "italic"
</STYLE>
```

To deploy the inset paragraph with the alternate properties, the container tag specifies both the `CLASS` and `ID` attributes:

```
<P CLASS="inset" ID="altInset">...</P>
```

The all keyword

For one more slice on the organization of styles, you can specify that a given style class is applicable to any tag in the document. In contrast, under normal circumstances, a class is restricted to the tags specified in the classes object definition. The `all` keyword defines a class that can be used for all tags in a document. For example, consider the following style definition:

```
<STYLE TYPE="text/javascript">
       classes.hotStuff.all.color = "red"
       classes.normal.p.margins("2em","3em","2em","3em")
       classes.normal.p.textAlign = "justify"
       classes.inset.p.margins("2em","10em","2em","10em")
       classes.inset.p.textAlign = "center"
       classes.inset.p.fontWeight = "bold"
</STYLE>
```

With this definition, the `hotStuff` class could be applied to any tag to turn its content red. But if I wanted to make the content of a `<DIV>` tag bold and center-aligned, I could not use the `inset` class, because that class is defined to be applied only to a `<P>` tag.

Be sure to separate in your mind Navigator's `all` keyword from Internet Explorer's `all` keyword. The former applies to style classes; the latter applies to document objects in general (in the context of the Internet Explorer 4 document object model).

Contextual styles

One final entry in Navigator's JavaScript Style Sheet vocabulary is the `contextual()` method. It provides yet another way to specify the application of a specific style. In this case, it defines what outer block tag surrounds a tag with a special style. For example, if I want all `<B>` tags inside `<P>` tags (but nowhere else) to display its text in red, I would define a contextual connection between the `<P>` and `<B>` tags as follows:

```
<STYLE TYPE="text/javascript">
      contextual(tags.p, tags.b).color = "red"
</STYLE>
```

Given the above style, a `<B>` tag set in a `<DIV>` or `<H1>` block would be displayed in the default color; but inside any `<P>` block, the color would be red.

Style Properties

If you use JavaScript Style Sheets, you must use the JavaScript syntax for setting style properties. For the most part, the property names are interCap versions of the CSS property names (which often include hyphens that JavaScript can't handle in identifiers). In-depth coverage of these properties and how to use them are within the scope of a volume dedicated to Dynamic HTML, not a JavaScript book, but I would be remiss if this book did not include at least a quick reference to the terminology. If the day should come that more of these properties are modifiable on the fly, then all of these properties (and the tags, classes, and ids objects) will become part of Navigator's document object model and move into the mainstream of JavaScript programming on Navigator.

Properties listed in this section are properties of the tags, classes, and ids objects. Many properties rely on specific settings, rather than numeric values. As a result, these items have a default value that governs the property if the property is not explicitly set in the style sheet.

In the property listings that follow, you will see several property value types listed as *size*. This usually means that the property must be set to a particular length that can be specified in several different units. When the content is designed for display on computer screen (as opposed to designed exclusively for printing), the unit is usually the pixel, abbreviated "px." Thus, a value to be set to 20 pixels would be entered as `20px`. Other supported unit measures are points, ems, picas, and x-height, entered as `pt`, `em`, `pi`, and `ex`, respectively.

As with HTML attributes that specify color, style properties for colors accept either Netscape color names or the standard RGB hexadecimal-triplet color designation. Note, however, that in Navigator 4, if you retrieve a color value, it may be returned as a decimal equivalent of the hexadecimal value. This makes it difficult to test for the current color of an object.

Block-level formatting properties

A block-level element is one that is normally rendered as a self-contained item, beginning its display on a new line and ending its display such that the next element appears on a new line after the block. For example, all the heading tags (for example, `<H1>`) are block-level elements, because the content between the start and end tags always starts on a new line, and forces a pseudo line break after the end. Properties that affect block-level rendering are shown in Table 42-1.

Block-level elements lend themselves to being surrounded by borders. They also can have margins, padding between borders and content, and other properties that befit a rectangular space on the page. When a block-level element property is one of a matched set, such as the top, left, bottom, and right edges of a block, JavaScript style sheets usually provide a single method that lets you set all four properties in one statement. For example, you can set the `borderWidth` property for any single side with the specific property name (such as

borderWidthTop), but to set all four at once, use the borderWidths() method, and fill in four parameters to set a block's complete border width specification:

```
classes.special.div.borderWidths("2px","3px","2px","3px")
```

Before using these methods, be sure you understand which parameter treats which side of the block. In the case of borders, the parameter sequence starts on the top edge and progresses clockwise around the rectangle.

Table 42-1
Block-level Style Properties

Property	***Default***	***Values***
align	none	left | right | none
backgroundColor	*browser default*	*colorName* | *RGB*
backgroundImage	*empty*	*URL*
borderBottomWidth	0	*size*
borderLeftWidth	0	*size*
borderRightWidth	0	*size*
borderTopWidth	0	*size*
borderWidths()	0,0,0,0	*valueTop*, *valueRight*, *valueBottom*, *valueLeft*
borderColor	none	*colorValue* | none
borderStyle	none	none | solid | 3D
clear	none	left | right | both | none
color	*browser default*	*colorName* | *RGB*
height	auto	*size* | auto
marginBottom	0	*size* | *percentage* | auto
marginLeft	0	*size* | *percentage* | auto
marginRight	0	*size* | *percentage* | auto
marginTop	0	*size* | *percentage* | auto
margins()	0,0,0,0	*valueTop*, *valueRight*, *valueBottom*, *valueLeft*
paddingBottom	0	*size* | *percentage*
paddingLeft	0	*size* | *percentage*
paddingRight	0	*size* | *percentage*
paddingTop	0	*size* | *percentage*
paddings()	0,0,0,0	*valueTop*, *valueRight*, *valueBottom*, *valueLeft*
width	auto	*size* | auto

Font and text properties

The next grouping, shown in Table 42-2, encompasses properties that control the look of fonts and text. These properties can be assigned to block-level and in-line elements.

To specify a value for the `fontFamily` property, you can name specific fonts or classes of fonts, each separated by a comma. The browser attempts to find a match for the font name on the user's system (the list of available fonts may be smaller than installed fonts, however). If no match occurs, the browser tries to use the next font specification in the font family list. For example, if you want to define Arial as the primary font and a generic serif font from the built-in list of generic font families (`serif`, `sans-serif`, `cursive`, `monospace`, and `fantasy`), define a tag style property like the following:

```
tags.p.fontFamily = "Arial, serif"
```

When a text property in Table 42-2 includes choices for a size or a percentage, the property may be influenced by related settings of a parent container. For example, if a paragraph tag has its `fontSize` property set to `large`, a style for an in-line tag inside that paragraph uses the parent container's settings as a starting point. Therefore, to increase the font size for an in-line item (say, a `<DIV>` tag) to 150 percent of the parent, define the style as follows:

```
tags.div.fontSize *= 1.5
```

With the multiply-by-value operator at work here, the property value acts as a percentage value over the value inherited from the parent. You can read more about the fine points of style inheritance at Netscape's online documentation for Dynamic HTML (`http://developer.netscape.com/library/documentation/-communicator/dynhtml/`).

Table 42-2
Font and Text Style Properties

Property	*Default*	*Values*
`fontFamily`	*browser default*	*fontFamilyList*
`fontSize`	medium	x-small | small | medium | large | x-large | xx-large | larger | smaller | [+/-] *integer* | *percentage*
`fontStyle`	normal	normal | italic | italic small-caps | oblique | oblique small-caps | small-caps
`fontWeight`	normal	normal | bold | bolder | lighter | 100 - 900
`lineHeight`	*browser default*	*number* | *size* | *percentage*
`textAlign`	*browser default*	left | right | center | justify
`textDecoration`	none	none | underline | overline | line-through | blink

(continued)

Table 42-2 *(continued)*

Property	***Default***	***Values***
textIndent	0	*size* | *percentage*
textTransform	none	none | capitalize | lowercase | uppercase
verticalAlign	baseline	baseline | sub | super | top | text-top | middle | bottom | text-bottom | percentage

Classification properties

One last group of properties, shown in Table 42-3, works more in the plumbing of style sheets, rather than impacting visual elements on a page. The first property, `display`, defines whether a tag should be treated as a block, inline, or list-item element. All HTML tags have default settings for this property, but you can override the default behavior, which may come in handy as you deploy generic tags such as <DIV> and <SPAN> for special purposes in your document.

The second property, `listStyleType`, gives you control over the way unordered and ordered lists display leading characters for their items. You assign this property to any tag whose display property is set to list-item (such as the <UL> and <OL> tags). And the third property, `whiteSpace`, defines how white space in the HTML source should be treated by the browser rendering engine. Normal processing collapses white space, but setting the property to `pre` is the same as surrounding a source in a <PRE> tag set.

Table 42-3
Classification Properties

Property	***Default***	***Values***
display	*HTML default*	block | inline | list-item | none
listStyleType	disc	disc | circle | square | decimal | lower-roman | upper-roman | lower-alpha | upper-alpha | none
whiteSpace	normal	normal | pre

Dynamic Positioning

In addition to supporting the Cascading Style Sheets-Positioning (CSS-P) standard, Navigator 4 also features its own implementation of a Dynamic HTML positioning system. This system is built around the layer object — generally created via the <LAYER> tag — which is so far unique to Navigator. Chapter 19 goes into great detail about the layer object and provides several example pages that help you learn how the object's properties, methods, and event handlers work.

In this chapter, I apply the <LAYER> object to the Dynamic HTML example shown for the first time as a cross-platform DHTML application in Chapter 41. Because the version in this chapter deals with only one platform, the amount of code is substantially less.

Navigator puzzle game overview

The Navigator-only version of the game relies on two HTML files and numerous image files. The main HTML file loads to reveal the play area (Figure 42-1). Content and HTML rendering instructions for a normally hidden panel of instructions are in the second HTML file. That second file is loaded as the source for a <LAYER> object defined in the main document. But, as you will see, even the secondary document contains JavaScript Style Sheets to help format the content as desired. Image files for this and the other two versions of the game are completely identical.

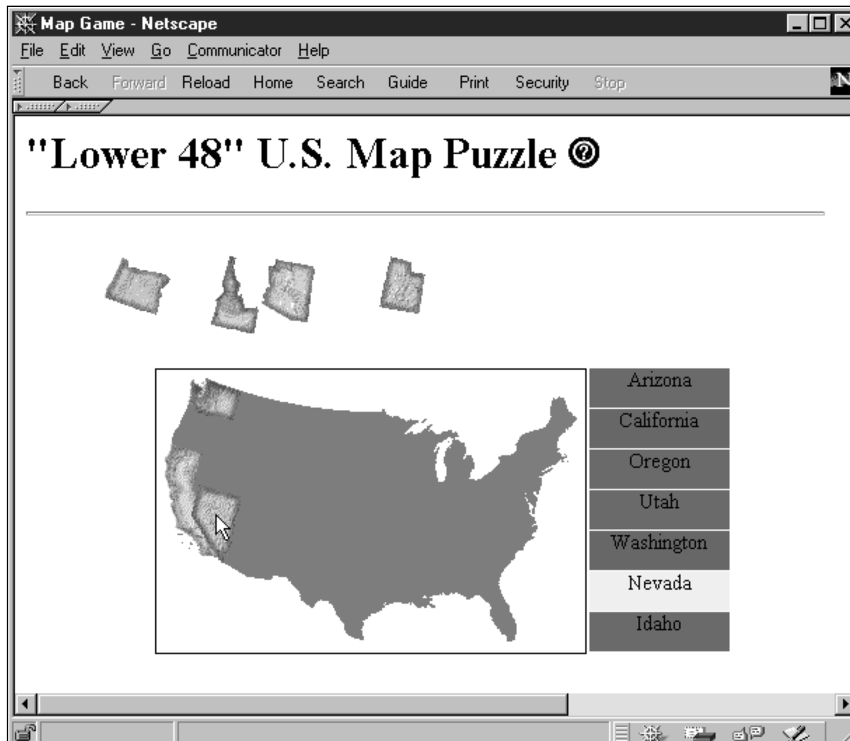

Figure 42-1: The Netscape-only version of the puzzle game (Image coutesy of Cartesia Software—www.map-art.com)

Structure of the main HTML document is straightforward. A large <SCRIPT> tag segment in the Head portion contains all the scripting for the main document (the instructions file has its own scripting). Document-level event capture is set for the main document to control the picking up, dragging, and release of state maps.

The main document

Listing 42-1 contains the entire source code for the main document. In most cases, the scripting code is identical to the Navigator portion of scripts in the cross-platform version. One large exception is that there is no external library of DHTML APIs, because the scripts can take direct advantage of Navigator's own vocabulary for layer properties and methods.

A big divergence between the versions appears in the usage of the <LAYER> tag throughout the Body portion of this chapter's version. Each positionable element is defined as a layer. Layer object properties that have the same names as JavaScript style properties are not the same items. For example, you cannot set the height and width of a layer object via the style sheet syntax in a <STYLE> tag elsewhere in the document.

For in-depth descriptions of the functions in the script and the structure of the layer objects, see the commentary associated with Listing 41-2.

Listing 42-1: The Main Document (NSmapgam.htm)

```
<HTML>
<HEAD><TITLE>Map Game</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// global variables
// click offsets
var offsetX = 0
var offsetY = 0
// info about clicked state map
var selectedObj
var selectedState = ""
var selectedStateIndex
var intervalID

// state object constructor
function state(abbrev, fullName, x, y) {
        this.abbrev = abbrev
        this.fullName = fullName
        // correct destination position
        this.x = x
        this.y = y
        this.done = false
}
// build states objects
var states = new Array()
states[0] = new state("ca", "California", 107, 234)
states[1] = new state("or", "Oregon", 107, 204)
states[2] = new state("wa", "Washington", 123, 188)
states[3] = new state("id", "Idaho", 148, 197)
states[4] = new state("az", "Arizona", 145, 285)
states[5] = new state("nv", "Nevada", 127, 241)
states[6] = new state("ut", "Utah", 155, 249)
```

```
// find out which state map layer is clicked on
function getSelectedMap(clickX, clickY) {
        var obj
        var testObj
        for (var i = states.length - 1; i >= 0; i--) {
           testObj = document.layers[states[i].abbrev + "map"]
           if ((clickX > testObj.left) && (clickX < testObj.left +
testObj.clip.width) && (clickY > testObj.top) && (clickY < testObj.top
+ testObj.clip.height)) {
              obj = testObj
              break
           }
        }
        selectedObj = obj
        if (selectedObj) {
           selectedStateLabel =
document.bgmap.document.layers[states[i].abbrev + "label"]
           selectedStateIndex = i
           // zoom the selected map to the frontmost layer
           selectedObj.zIndex = 100
        }
}
// handle drag
function dragIt(e) {
        if (selectedObj) {
           selectedObj.moveTo(e.pageX - offsetX, e.pageY - offsetY)
        }
}
// turn click mode on and off
function toggleEngage(e) {
        if (selectedObj) {
           release(e)
        } else {
           engage(e)
        }
}
// turn click mode on
function engage(e) {
        // set global selected values
        getSelectedMap(e.pageX, e.pageY)
        if (selectedObj) {
           // store click offsets within layer and save to globals
           offsetX = e.pageX - selectedObj.left
           offsetY = e.pageY - selectedObj.top
           // turn corresponding label background to yellow
           selectedStateLabel.bgColor = "yellow"
        }
}
// handle mouse button release
function release(e) {
        if (selectedObj) {
           // see if state has been dragged to within 4 pixels of
destination
```

(continued)

Listing 42-1 *(continued)*

```
            if (onTarget(e)) {
               // set label to green
               selectedStateLabel.bgColor = "green"
               // and state object's done property to true
               states[selectedStateIndex].done = true
               // see if you're done and flash the labels
               if (isDone()) {
                  document.congrats.visibility="visible"
               }
            } else {
               // otherwise revert label to red
               selectedStateLabel.bgColor = "red"
               states[selectedStateIndex].done = false
               document.congrats.visibility="hidden"
            }
            // set object z-order to bottom
            selectedObj.zIndex = 0
            selectedObj = null
            selectedState = ""
         }
}
// test whether dragged map is near destination
function onTarget(e) {
         // get destination coords from states object
         var x = states[selectedStateIndex].x
         var y = states[selectedStateIndex].y
         // get current location
         var objX = selectedObj.pageX
         var objY = selectedObj.pageY
         // see if object is within 4 pixels of destination
         if ((objX >= x-2 && objX <= x+2) && (objY >= y-2 && objY <=
y+2)) {
            // snap object to destination
            selectedObj.moveTo(x,y)
            return true
         }
         return false
}
// test if all states have been placed
function isDone() {
         for (var i = 0; i < states.length; i++) {
            if (!states[i].done) {
               return false
            }
         }
         return true
}
function showHelp() {
         var help = document.help
         help.visibility = "show"
```

```
        help.zIndex = 1000
        intervalID = setInterval(moveHelp,1)
}
function moveHelp(obj) {
        var help = document.help
        help.moveBy(-5,0)
        if (help.pageX <= (window.innerWidth/2) - 150) {
           clearInterval(intervalID)
        }
}
function setEvents() {
        document.captureEvents(Event.MOUSEDOWN | Event.MOUSEMOVE)
        document.onMouseDown = toggleEngage
        document.onMouseMove = dragIt
}
</SCRIPT>
</HEAD>
<BODY onLoad="setEvents()">
<H1>"Lower 48" U.S. Map Puzzle <A HREF="javascript:void
showHelp()" onMouseOver="status='Show help panel....';return true"
onMouseOut="status='';return true"><IMG SRC="info.gif" HEIGHT=22
WIDTH=22 BORDER=0></A></H1>
<HR>
<LAYER ID="help" TOP=80 LEFT=&{window.innerWidth}; WIDTH=300
VISIBILITY="HIDDEN" SRC="instrux.htm"></LAYER>

<LAYER ID="bgmap" TOP=180 LEFT=100><IMG SRC="us11.gif" WIDTH=306
HEIGHT=202 BORDER=1>
<LAYER ID="azlabel" TOP=0 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Arizona</CENTER></LAYER>
<LAYER ID="calabel" TOP=29 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>California</CENTER></LAYER>
<LAYER ID="orlabel" TOP=58 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Oregon</CENTER></LAYER>
<LAYER ID="utlabel" TOP=87 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Utah</CENTER></LAYER>
<LAYER ID="walabel" TOP=116 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Washington</CENTER></LAYER>
<LAYER ID="nvlabel" TOP=145 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Nevada</CENTER></LAYER>
<LAYER ID="idlabel" TOP=174 LEFT=310 BGCOLOR="red" WIDTH=100
HEIGHT=28><CENTER>Idaho</CENTER></LAYER>
</LAYER>

<LAYER ID="congrats" TOP=100 LEFT=20 VISIBILITY="HIDDEN"><H1><FONT
COLOR="red">Congratulations!</FONT></H1></LAYER>
<LAYER ID="camap" TOP=100 LEFT=20><IMG SRC="ca.gif" WIDTH=47 HEIGHT=82
BORDER=0></LAYER>
<LAYER ID="ormap" TOP=100 LEFT=60><IMG SRC="or.gif" WIDTH=57 HEIGHT=45
BORDER=0></LAYER>
<LAYER ID="wamap" TOP=100 LEFT=100><IMG SRC="wa.gif" WIDTH=38 HEIGHT=29
BORDER=0></LAYER>
```

(continued)

Listing 42-1 *(continued)*

```
<LAYER ID="idmap" TOP=100 LEFT=140><IMG SRC="id.gif" WIDTH=34 HEIGHT=55
BORDER=0></LAYER>
<LAYER ID="azmap" TOP=100 LEFT=180><IMG SRC="az.gif" WIDTH=38 HEIGHT=45
BORDER=0></LAYER>
<LAYER ID="nvmap" TOP=100 LEFT=220><IMG SRC="nv.gif" WIDTH=35 HEIGHT=56
BORDER=0></LAYER>
<LAYER ID="utmap" TOP=100 LEFT=260><IMG SRC="ut.gif" WIDTH=33 HEIGHT=41
BORDER=0></LAYER>

<SCRIPT LANGUAGE="JavaScript">
</SCRIPT>
</BODY>
</HTML>
```

The help panel

Content for the flying help panel comes from the second file, instrux.htm. The source for that document is shown in Listing 42-2. Although quite simple as HTML goes, this document benefits from a couple style rules. Without setting the `marginTop` property of the first paragraph or the `marginRight` property of the ordered list entries, the text would be positioned too close to the edges of the panel. Other style properties could also be used to make the same adjustments. You can see the results in Figure 42-2.

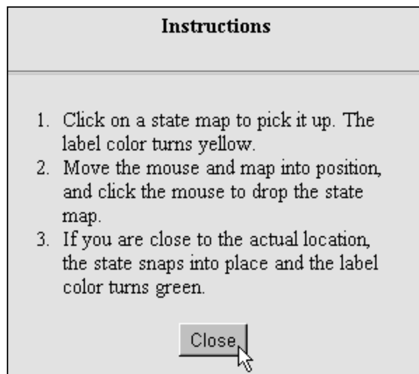

Figure 42-2: The help panel at rest

An advantage of having a layer's content loaded as a separate HTML document comes home in this example. All event handling and scripting related to this positionable element is encapsulated in the external document. Nothing from this document clutters the source code of the main document, except for the animation routine. Moreover, the content can be reused in another document by having another `<LAYER>` tag load it in.

Listing 42-2: The Help Panel Document (instrux.htm)

```
<HTML>
<HEAD>
<STYLE TYPE="text/javascript">
tags.P.marginTop = 5
tags.OL.marginRight = 20
</STYLE>
<SCRIPT LANGUAGE="JavaScript">
window.captureEvents(Event.CLICK)
window.onclick = hideMe
function hideMe() {
        window.document.help.visibility = "hidden"
        window.document.help.moveTo(window.innerWidth,120)
}
</SCRIPT>
</HEAD>
<BODY BGCOLOR="#98FB98">
<P><CENTER><B>Instructions</B></CENTER></P>
<HR COLOR="seagreen">
<OL>
<LI>Click on a state map to pick it up. The label color turns yellow.
<LI>Move the mouse and map into position, and click the mouse to drop
the state map.
<LI>If you are close to the actual location, the state snaps into
place and the label color turns green.
</OL>
<FORM>
<CENTER>
<INPUT TYPE="button" VALUE="Close" onClick="">
</CENTER>
</FORM>
</BODY>
</HTML>
```

Lessons learned

Because I am comfortable with Netscape's document object model and the hierarchy of nested objects such as frames, the experience of building this page with the <LAYER> tag and its scriptable pieces seemed like a natural extension to my previous knowledge. Using separate HTML files as building blocks to a complex document also has its appeal in a number of application scenarios. At the same time, it becomes clear that JavaScript Style Sheets and positionable elements are two distinct components of Navigator Dynamic HTML. There is very little overlap in the syntax or objects in the implementation of Navigator 4. Even so, if my deployment environment were guaranteed to be Navigator 4 only, I would build the application with the Netscape-specific extensions.