CHAPTER 41

Scripting Cross-Platform Dynamic HTML

✦ ✦ ✦ ✦

In This Chapter

Introduction to Dynamic HTML

The common denominator of DHTML functionality across browsers

An interactive map puzzle written for cross-platform deployment

✦ ✦ ✦ ✦

Level 4 browsers — Navigator 4 and Internet Explorer 4 — incorporate some of the latest World Wide Web technologies for display and control over Web page content. Lumped together under the heading of Dynamic HTML (DHTML), these technologies dramatically extend the simple formatting of standard HTML that page authors have used for years. As the installed base of level 4 browsers grows, the competition among application authors will drive the desire to provide more engaging and satisfying applications with little added cost in download time for the user.

A lot of what the user gets with DHTML has previously been accomplished only via Java applets and plug-ins, such as ShockWave. Not that DHTML will eliminate these technologies from the Web author's arsenal (DHTML doesn't do sound or video, for example), but because DHTML can accomplish much more of what authors look for in assembling page content and layout without the long downloads of applets or plug-in content, it becomes an attractive way for nonprogrammers to spice up Web applications.

Perhaps categorizing DHTML authors as "nonprogrammers" is not quite right. DHTML also adds significantly to the vocabulary required to incorporate dynamic content into pages. Suddenly HTML becomes a lot more programming than simply adding tags to existing content. And if you want to do dynamic positioning of elements, be prepared to put your JavaScript skills to use.

What is DHTML?

You can practically find as many definitions of Dynamic HTML as there are people to ask. This is especially true if you ask Netscape and Microsoft. Each company defines DHTML in terms of the support its browser has for a variety of technologies. My definition covers a broad range, because

DHTML is not really any one "thing." Instead it is an amalgam of several technologies, each of which has a standards effort in varying stages of readiness. The key technologies are as follows: cascading Style Sheets (CSS1); cascading Style Sheets-Positioning (CSS-P); document Object Model (DOM); and client-side scripting.

To this list I also admit recent advances in downloadable font technology and Extensible Markup Language (XML), the latter opening the door to author-generated, page-specific HTML extensions that don't rely on standards bodies or browser support. Both of these are currently beyond the scope of a JavaScript-centered book.

Cascading Style Sheets (CSS1)

The "1" in the acronym for Cascading Style Sheets stands for "Level 1." As this book goes to press, a working draft for Level 2 (CSS2) has been released for public comment.

A style sheet defines a display characteristic for a document element. Any element contained by an HTML tag can be directed to adhere to one or more settings pertaining to the display of that content. For example, a paragraph of text (delimited by a `<P>...</P>` tag pair) might have a style sheet attached to it such that the right margin is wider than normal, its text color is blue, and a colorful border surrounds the entire chunk of text.

The combination of an HTML tag identifier and its style setting is called a *rule*. Components of a rule are the *selector* (the tag identifier, such as `H1`) and the *declaration*. A declaration itself has two components: A *property* name and a *value*, separated by a colon. For example, a common way to define a style sheet is inside a `<STYLE>` tag, as follows:

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

In this definition, the type of style sheet is specified as a Cascading Style Sheet type whose style specification is straight text. The rule establishes that all blocks surrounded by `<P>...</P>` tag pairs (the `P` selector) will have their text colored red. Here, the property being specified is `color`, and its value red. Notice that the definition is inside curly braces and that the property and its value are separated by a colon. This may take some getting used to after defining attribute-value pairs in HTML as separated by an equals sign. If you want to add another declaration to the same rule, you separate the first from the second by a semicolon, as follows:

```
<STYLE TYPE="text/css">
       P {color:red; font-size:14pt}
</STYLE>
```

The above style sheet tells the browser that every `<P>`-tagged block of text in the document (below this style definition in the document) is to be displayed in red at 14 points.

The CSS1 recommendation offers several ways to incorporate style sheets into a document, including linking in external documents that contain style sheet rules (very much like linking in an external .js script library). Other parts of the specification detail how you override a blanket rule with a tag-specific rule. For example, if a document were governed by the preceding paragraph rule, an individual paragraph in the document can have its own color with the following beginning tag:

```
<P STYLE="color:blue">
```

By changing only one property, the prevailing font size from the earlier rule applies to this paragraph as well.

Even more sophisticated parts of the CSS1 specification allow for creating subsets (classes) of each style. For example, if your document required several instances of two paragraph styles, you can define `P`-level rules for each, assigning each style a class name. Then, in the actual `<P>` tag where each document exists, specify the `CLASS` attribute so the browser knows which `P`-level rule to apply to the ensuing paragraph.

As mentioned earlier in the book, industry standards for Cascading Style Sheets is under the aegis of the W3C organization. As of this writing, CSS1 is a published recommendation, available at `http://www.w3.org/pub/WWW/TR/REC-CSS1`.

Both Navigator 4 and Internet Explorer 4 support CSS1 (Internet Explorer 3 also had some support for CSS), although neither implementation is flawless. While CSS1 is not necessarily dynamic (that is, once you lay out a page according to a style, that's as far as the specification goes), Internet Explorer 4 makes it truly dynamic. As explained in detail in Chapter 43, the content and style of each HTML tag can be changed on the fly from a script. Changes to the page that require rerendering of the page are handled well, with very fast reflowing of content. In this regard, the rendering engine of Internet Explorer 4 is superior to that of Navigator 4 (at least as of Navigator Version 4.04).

Netscape added another type of style sheet system to Navigator 4. Called JavaScript Style Sheets (JSS), this style sheet methodology's syntax more closely resembles JavaScript. It is covered in detail in Chapter 42.

Cascading Style Sheets-Positioning (CSS-P)

An entirely separate standards effort is assigned to the positioning of HTML elements on a page. The goal is to extend the syntactical framework of CSS1 so that items can be precisely positioned on a page. Without precise positioning of content, Web publishing is extremely crude compared to the current state of the art, desktop publishing (DTP). For content providers with experience in DTP, adapting carefully designed content to the Web is often a frustrating — and certainly limiting — experience.

Positioning encompasses more than simply defining a point on the page where an element is to start rendering itself. Built into the specification are properties for whether an element is visible or hidden and whether one element that overlaps another is visibly closer to the user's eye than elements behind it. With a syntax in place for setting these properties as the page loads, it is a small extension to making these properties visible to JavaScript. When that happens, the properties become truly dynamic, allowing for moving an element across the screen like an animation, and hiding or showing elements as needed.

Although the CSS-P recommendation is only at the working draft stage at the time of release of Navigator 4 and Internet Explorer 4, both browsers implement

CSS-P, albeit not in exactly the same way. You can view the W3C draft at `http://www.w3.org/pub/WWW/TR/WD-positioning-970131.html`. By the time you read this, it is possible that a more recent draft or final recommendation will have been published by W3C.

The truly dynamic nature of CSS-P will likely attract scripters. Most of the discussion in this and the next two chapters focuses on CSS-P, rather than the less-dynamic CSS1. Both, however, have a lot in common with basic property and value pair syntax and structure.

Document Object Model (DOM)

Perhaps the most important piece of the DHTML puzzle is a specification for the Document Object Model. I assign more weight to the DOM than to, say, CSS-P, because it is the definition of a standard DOM that scripters will rely on for cross-platform consistency in their scripts. An inconsistency among document object models for Internet Explorer 3 and Navigator 3 — particularly the lack of an Image object in Internet Explorer 3 — caused no little grief among scripters who developed mouse rollover scripts in Navigator 3, only to have them cause script errors for Internet Explorer 3 users.

Given the split evolution of document object models in Navigator and Internet Explorer (especially with their divergence in level 4 versions), a recognized DOM standard for scriptable objects will be difficult to achieve. The more each platform has invested in the installed base of its existing document object model, the more difficult it will be to reach a consensus, which may require rolling back features to reach a scriptable common denominator.

Of all the standards efforts described in this chapter, the Document Object Model is the least far along in its development. As of this writing, the most recent document is the working draft of HTML-specific syntax. The W3C has assumed responsibility for this standard. The earliest its results will reach browsers is perhaps the level 5 versions of Navigator and Internet Explorer.

Client-side scripting

Scripting languages such as JavaScript provide the bridge between an author’s ideas for interactivity and the controllable elements on the page. I specify the broader “client-side scripting” rather than JavaScript here, because future HTML standards that accommodate scripting are leaving room for any language that a browser wishes to support. Navigator, of course, supports only JavaScript. Internet Explorer, on the other hand, supports its own version of JavaScript (JScript) and VBScript (a derivative of Visual Basic). Because of its cross-platform support, JavaScript is the predominant client-side scripting language today.

As discussed in Chapter 2, the core language aspects of JavaScript have been standardized into a language called ECMAScript (after the ECMA standards body that oversaw the effort). This specification (ECMA-262) is for the core language that does not involve document objects, but rather language definitions for constructs such as variables, data types, operators, and so on. Because Dynamic HTML is all about scripting page elements (document objects), the core language serves primarily as essential support for the entirety of JavaScript that deals with the object model as well.

Cross-Platform DHTML Challenges

Despite all the standards efforts for DHTML technologies, both Netscape and Microsoft develop their own, frequently mutually exclusive, extensions to the standards. In the case of CSS-P, both browsers let scripts control identical positioning characteristics, but differences in the document object models render any chance of writing one script that works in both browsers virtually impossible unless you are familiar with the idiosyncracies of each implementation.

To help you understand the challenges that you'll face in creating DHTML pages for both browsers, I will introduce you to the implementation of positionable elements — I like to call them *positionable thingies*, or *PTs* for short — in both Navigator and Internet Explorer. Until you know how to create JavaScript references to PTs in both platforms, you will never be able to write one script for all browsers.

Netscape extensions — the layer

Due to the timing of the release of Navigator 4, some of the standards built into the later-shipping Internet Explorer 4 were not finalized in time to be in Navigator 4. Moreover, Netscape had proposed a new HTML tag to account for positionable thingies. That tag is the `<LAYER>` tag. The W3C organization that oversees the HTML specification and many of the DHTML specifications did not adopt the `<LAYER>` tag, but Netscape had already built its flavor of DHTML around this object (see Chapter 19). In the ever-changing political climate of the browser wars, Netscape (as of this writing, anyway) is downplaying the term "layer," even though it is plainly part of the tag and scripted object vocabulary. Where this current thinking will lead is anyone's guess.

In the meantime, if you want to script a PT in Navigator, you must use the layer object vocabulary, even if you don't use a `<LAYER>` tag to create a PT. As discussed in detail in Chapter 19, the layer object features a rich set of properties and methods that let you adjust the position, visibility, and z-order of objects on a page. In the HTML for a page containing a layer, the content of a layer can be embedded in the same document or loaded from an external HTML file. Once the page is loaded, the base document contains its own content, plus any documents loaded via the `<LAYER>` tag.

Object references to layers mimic the same hierarchical approach that you use for frames in a frameset. The more deeply nested an object, the longer the reference. You must also pay attention to the layer containing the script that needs to communicate with another layer (or one of its elements) and traverse the layer hierarchy in the reference (for example, when using the `parentLayer` property).

An important point to remember about a layer is that its content is associated with a document object contained by that layer. Therefore, the layer itself has properties and methods that affect its "layer-ness"; but if you want scripted access to the content of the layer (for example, a form element value), you must refer to the layer's document when you assemble your scripted reference.

Because of the hierarchical nature of layers in Navigator, a reference to a layer and its content must represent the hierarchy. No global view of all layer objects in the document exists. Therefore, even if you assign unique names to each layer object (as you should anyway), that won't help the browser locate the object without the complete hierarchical map to the object in the reference.

If you are comfortable with the frame referencing mechanism in JavaScript (in both Navigator and Internet Explorer), you may well feel comfortable applying the same methodology to layer objects in Navigator 4. It is also often convenient to drop in an external HTML document at a location within the main document. As a result, Netscape's layer object provides a good mental model for many scripters.

Microsoft extensions—style objects

Internet Explorer 4 ignores the `<LAYER>` tag, so nothing you define in a document with that tag will become a PT in Internet Explorer. Instead, Microsoft turns the PT concept on its side. You create a PT in Internet Explorer by virtue of including one special property-value pair in a style definition (either in a `<STYLE>` tag or in the `STYLE` attribute of any container tag). That property is the `position` property, and it has two possible values: `relative` and `absolute`. This is not specifically an Internet Explorer invention, but rather is spelled out in the CSS-P specification.

Microsoft's spin on the matter comes in the way scripts reference these thingies. Rather than referring to them as physical layers you can move around, hide, show, and stack in different orders, Internet Explorer refers to these objects as `styles`. A style is an extremely powerful object in the Internet Explorer 4 document object model. It has dozens of properties, each reflecting the properties that can be set according to the CSS1 and CSS-P recommendations. A style also contains content in the form of HTML that can be modified on the fly after the document has loaded.

All of a style object's controls for adjusting positioning characteristics are properties. Unlike the Navigator layer, a style object has no methods for positioning.

Finding the Common Denominator

If you intend to deploy a Dynamic HTML page that works with both Navigator and Internet Explorer, you need to know how to create a common denominator of objects that both browsers recognize. Citing its adherence to the CSS1 and CSS-P recommendations, Microsoft recognizes only one kind of positionable object: its style object.

Netscape, however, has one extra trick up its sleeve. While Navigator only recognizes layers, it is more lenient about how you define a layer object. In addition to observing the `<LAYER>` tag as a layer object maker (and the `new Layer()` constructor), any object defined as a CSS-P object is automatically recognized as a layer and can be referenced with the Navigator layer syntax.

Creating PTs

The common denominator for creating a PT is the CSS-P syntax that defines a style with the `position` property. Therefore, you can use the `<STYLE>` tag that created named items, as follows:

```
<HTML>
```

```
<HEAD>
<STYLE TYPE="text/css">
       #thingy1Name {position:absolute}
       #thingy2Name {position:absolute}
</STYLE>
</HEAD>
```

Later in the document, you assign the item name to the `ID` attribute of a container tag:

```
<DIV ID="thingy1Name">...</DIV>
```

All position properties are applied to the content of the container before it is rendered on the page. As an alternate syntax, you define a `STYLE` attribute for a container tag as follows:

```
<DIV ID="thingy1Name" STYLE="position:absolute">...</DIV>
```

For either approach, you can specify additional positioning properties as you like. For example, a positionable thingy named `controlPanel` would be positioned at a top-left location of 100, 20 and initially hidden as the page loads, from the following specification:

```
#controlPanel {position:absolute; top:20; left:100; visibility:hidden}
```

Table 41-1 shows the positionable object properties you can use on both platforms. Because this is part of the CSS-P recommendation, the syntax is identical for both platforms.

Table 41-1
CSS-P Rule Properties and Values

Property	***Values***
`position`	Absolute | relative
`left`	Pixels relative to the containing element
`top`	Pixels relative to the containing element
`visibility`	Visible | hidden | inherit
`z-index`	Layer position in stack (integer)

Note

In Navigator 4, you may encounter a problem with a PT defined in the CSS-P syntax and later shown, moved, and hidden by script control. For unknown reasons (that is, bugs), after scripts move and hide the object, events are prevented from reaching existing objects. To work around the problem, you can use JavaScript to `document.write()` the portion of the document affected by the problem. For Navigator users, `document.write()` a `<LAYER>` tag; for Internet Explorer 4 users, `document.write()` the CSS-P syntax. Later in this chapter I demonstrate how to execute this methodology.

References to positionable thingies

Once a PT is created, your script may need to read or modify one of its properties. You need to watch for two factors. The first is the way you reference the thingy; the other is the property or method you use to extract or adjust a setting.

Referencing the object is the first cross-platform hurdle. Recall that Navigator uses the layer object as its PT, while Internet Explorer uses the style object. References to layer objects must also take into account the hierarchy of nested layers, if any. Styles, on the other hand, can be accessed globally in a window or frame.

Navigator layer references

Consider a script in the main document that wants to adjust a position property of a layer created at that document level. The reference format would be

```
document.layerName
```

Layers are also stored as arrays of the document object, so you can use array indexing and naming techniques to access that layer object:

```
document.layers[0]
document.layers["\"]
```

To access content of that layer, the reference must include the document inside the layer that contains the content. For example, to access the value of a text box in a form in a layer would require a reference with the following format:

```
document.layerName.document.formName.textBoxName.value
```

When a layer is nested inside another layer, the reference gets even longer. The traversal of the object hierarchy always goes through a document and a layer, because only a document can contain a layer, and every layer contains a document. Therefore, it would not be unfeasible to see a reference format such as the following:

```
document.layerName1.document.layerName2.document.formName.textBoxName.
value
```

For more details on referencing layer objects from a variety of locations in a layer hierarchy see Chapter 19.

Internet Explorer style references

Internet Explorer 4's document object model provides a way to transcend any hierarchical structure of nested style objects with a document property called `all`. A reference to `document.all` exposes all named objects in the document. To that reference beginning, you add the name of the object defined in the document, plus a further reference to the style property, according to the following format:

```
document.all.objectName.style
```

This `document.all` property exposes all kinds of objects. Therefore, to access the same text box value described previously in Internet Explorer syntax, the reference format would be

```
document.all.formName.textBoxName.value
```

Of course this means that you need to assign unique names to each addressable object, but you should be doing that anyway. This format also simplifies writing generalizable functions that need to work with style objects.

Property name incompatibilities

While the names for position properties are the same in both platforms, scripted access to them varies in a couple of key instances. Table 41-2 shows the corresponding scripted position-related property names for Navigator and Internet Explorer.

Table 41-2
CSS-P Scripted Object Properties Names

Navigator Layer Property	*Internet Explorer Style Property*
`left`	`pixelLeft`
`top`	`pixelTop`
`visibility`	`visibility`
`zIndex`	`zIndex`

The Internet Explorer style object does have `left` and `top` properties, but these values return the unit along with the coordinate value (for example, `20px`) bundled together as a string value. If you want to shift the position along the horizontal or vertical axis by adding or subtracting some number of pixels from the current position, you need the `pixelLeft` and `pixelTop` properties, which return integer values for the coordinates.

Also note in Table 41-2 that the property name for setting the stacking order of objects is `zIndex`, while the HTML style property is named `z-index`. Many CSS property names are hyphenated words. Unfortunately for compatibility, JavaScript does not allow hyphens in identifiers. Therefore, property names usually include the components of the CSS property names, but assembled in interCap format. This is true throughout Microsoft's extensive object properties that reflect CSS1 style properties. Navigator 4 does not have as thorough support for such properties. In some cases, Navigator even diverges from the CSS convention, instead following the scripting and HTML convention. For example, setting the background color of a positionable thingy requires setting the `background-color` property. Internet Explorer 4's scripted access to that property is `backgroundColor`. But Navigator follows the convention from its object model by using the `bgColor` property (the same as in `document.bgColor`).

About methods

Navigator defines a number of methods for its layer object (see Chapter 19). Several of these facilitate repositioning an object to a specific coordinate location or shifting an object by a number of pixels along each axis. Internet Explorer offers no equivalent methods for the style object. Instead, you set the `pixelLeft` and `pixelTop` properties.

These properties, as mentioned previously, have analogs in Navigator (`left` and `top`), so no matter how you plan to reposition an object in a cross-platform page, you will have to create branches in your code to accommodate each browser type. I personally find the Navigator methods convenient, since a single method call handles both axes.

Working Around Incompatibilities

The bottom line on incompatibilities is that you have to accommodate incompatible object references and occasionally incompatible property names. Scripting gives you several alternatives to working your way around these potential problems.

Of course, you are not obligated to create one page that works on both platforms. You are free to create a version that is optimized for each browser, and let some of the redirection techniques discussed in Chapter 13 automatically navigate users to the page for their browsers. To give you an idea of what such optimized designs might look like, the primary application examples in Chapters 42 and 43 are browser-specific versions of the cross-platform application assembled in this chapter.

In the rest of this section, I present several code examples of ways to accommodate both browsers in a single document. The three basic techniques are inline script branching, something I call *platform equivalency*, and custom APIs (application program interfaces).

Inline branching

Before you can begin to write code that creates branches for each browser, you should define two global variables at the top of the page that act as Boolean flags for your `if...else` constructions later. Therefore, at the first opportunity for a `<SCRIPT>` tag in a page, include the following code fragment to set flags named `isNav4` and `isIE4`:

```
var isNav4, isIE4
if (navigator.appVersion.charAt(0) == "4") {
       if (navigator.appName == "Netscape") {
           isNav4 = true
       } else if (navigator.appVersion.indexOf("MSIE") != -1) {
           isIE4 = true
       }
}
```

Version checking here is quite specific. First of all, it intentionally limits access to browsers whose versions come back as Version 4. Elsewhere in this book, I recommend making such conditions based on whether an integer representation of the version is the same or greater than a minimum level. This would accommodate the next generation of browser without having to modify the code. In this instance, however, I decided to limit the access only to Version 4. I would hope that future versions would be backward compatible to the code written for level 4 browsers, but DHTML is volatile enough that I limit access to level 4 browsers until I get a peek at the next generation.

Another aspect of the flag-setting script I should mention is that the example provides no escape route for browsers that aren't level 4 or aren't either Navigator or Internet Explorer (should there be a level 4 browser from another brand). In a production environment, I would either prefilter access to the page or redirect ill-equipped users to a page that explains why they can't view the page. In the structure of the above script, redirection would have to be made in two places, as follows:

```
var isNav4, isIE4
if (navigator.appVersion.charAt(0) == "4") {
       if (navigator.appName == "Netscape") {
          isNav4 = true
       } else if (navigator.appVersion.indexOf("MSIE") != -1) {
          isIE4 = true
       } else {
          location = "sorry.html"
       }
} else {
       location = sorry.html"
}
```

Later in this chapter, I discuss the issue of designing DHTML pages that degrade gracefully in pre-DHTML browsers.

With the global variables defined in the document, you can use them as condition values in branching statements that address an object according to the reference appropriate for each platform. For example, to change the `visibility` property of an object named `instructions`, you would use the flags as follows:

```
if (isNav4) {
       document.instructions.visibility = "hidden"
} else {
       document.all.instructions.style.visibility = "hidden"
}
```

Platform equivalency

Another technique attempts to limit the concern for the different ways each platform refers to a PT. If you examine the formats for each platform's object references, you see that both formats contain a reference to the document and to the object name. Internet Explorer 4's syntax also includes property words such as `all` and `style`. If you assign these extra property names to variables for Internet Explorer 4 and leave those variables as empty strings for Navigator 4, you can assemble an object reference for both platforms in one statement.

To begin using this technique, set two global variables that store reference components for the scope (`all` in Internet Explorer 4) and the style object (`style` in Internet Explorer 4):

```
var range = ""
var styleObj = ""
if (navigator.appVersion.charAt(0) == "4") {
       if (navigator.appVersion.indexOf("MSIE") != -1) {
          range = "all."
          styleObj = ".style"
       }
}
```

From this point, you can assemble an object reference with the help of the JavaScript `eval()` function, as follows:

```
var instrux = eval("document." + range + "instructions" + styleObj)
instrux.visibility = "hidden"
```

Or, you can use the `eval()` function to handle the entire property assignment in one statement, as follows:

```
eval("document." + range + "instructions" + styleObj + ".visibility =
'hidden'")
```

If your page does not have a lot of objects that your scripts will be adjusting, you can use this platform equivalency approach to create global variables holding references to your positionable objects at load time (triggered by the `onLoad=` event handler so that all objects exist and can be referenced by the `eval()` function). Then use those variables for object references throughout the scripts.

Unfortunately, the platform equivalency methodology breaks down when a Navigator layer object is nested inside another layer. The platform equivalency formulas assume that each object is directly addressable from the outermost document object. If your objects have a variety of nested locations, you can use either the inline branching method described earlier, or batch-assign objects to global variables at load time using platform branching techniques along the lines of the following example:

```
var instrux
function initObjectVars() {
        if (isNav4) {
            instrux = document.outerLayer.document.instructions
        } else {
            instrux = document.all.instructions.style
        }
}
```

Once the variable contains a valid reference to the object for the current platform, your scripts can treat the object without further concern for platform when addressing properties that have the same name in both platforms:

```
instrux.visibility = "hidden"
```

But when properties are different for each platform, you might want to examine the custom API approach.

Custom APIs

A JavaScript custom API is a function you design that acts as an intermediary between your scripts and other scriptable entities. Ideally, an API simplifies access to or control of other entities. In the context of designing a cross-platform CSS-P page, an API can offer a single function that smooths over the differences in object references and/or property names between platforms. Your custom function provides a single access point that is consistent across both platforms. In essence, you are creating your own metavocabulary for methods and property settings.

Let me begin with an example that needs to handle only the differences in object referencing. The example assumes that some of the global variables described earlier in other techniques (the browser version flags at the minimum) have been defined in the document's scripts. This API provides a function for setting the `zIndex` property of an object whose name is passed as a parameter:

```
function setZIndex(objectName, zOrder) {
        var theObj
        if (isNav4) {
           theObj = eval("document." + objectName)
        } else {
           theObj = eval("document.all." + objectName + ".style")
        }
        if (theObj) {
           theObj.zIndex = zOrder
        }
}
```

With this function in place, a script can invoke the `setZIndex()` function and not have to worry about platform. Since this API can be reused in all your documents (or, better yet, linked in as an external library), you can make it part of your DHTML scripting vocabulary.

The preceding example works only for PTs that are not nested in Navigator. If the page uses nested layers, you can reconfigure the API to accept either an object name — which assumes the object is not nested — or a positionable element object reference — which assumes nothing about its location:

```
function setZIndex(obj, zOrder) {
        var theObj
        if (typeof obj == "string") {
           theObj = eval("document." + range + obj + styleObj)
        } else {
           theObj = obj
        }
        theObj.zIndex = zOrder
}
```

In the reconfigured API, the parameters are the same, but the function takes advantage of JavaScript's loose data typing by accepting either a string or object reference as the first parameter. If the parameter is a string, then a valid object reference is built with the help of the platform equivalency methodology. Otherwise, the object reference parameter is processed as-is. When the function is passed an object reference, this API assumes that the script invoking this function has already assigned a valid positionable element object reference to the parameter being passed.

As one more example, the next API function offers an interface to incompatible ways of adjusting the location of a positionable thingy. For Navigator, it uses the `layer.moveTo()` method; for Internet Explorer, it sets the `pixelLeft` and `pixelTop` properties:

```
function shiftTo(obj, x, y) {
        var theObj
        if (typeof obj == "string") {
           theObj = eval("document." + range + obj + styleObj)
        } else {
            theObj = obj
        }
        if (isNav4) {
            theObj.moveTo(x,y)
        }
        if (isIE4) {
            theObj.style.pixelLeft = x
            theObj.style.pixelTop = y
        }
}
```

Both ways of setting an object’s location require x and y coordinates, but the values are applied differently. Therefore, the script invoking this API still doesn’t concern itself with the platform.

Handling Non-DHTML Browsers

An important question to ask yourself as you embark on a DHTML-enhanced page is how you intend to treat visitors whose browsers aren’t up to the task. In many respects the problem is similar to the problem of treating nonscriptable browsers when your page relies on scripting (see Chapter 13).

The moment your page uses DHTML to position an element, you must remember that non-DHTML browsers display the content according to the traditional HTML rendering rules. No elements are allowed to overlap. Any block-level tag is rendered at the left margin of the page, unless some other non-DHTML alignment tag is at work. This goes for elements that are DHTML-positioned to sit off screen (perhaps with a clickable tab) until called by the user. An element defined as being hidden in DHTML will be visible. In most cases, your carefully designed DHTML page will look terrible.

However, a page that does not use too radical a layout strategy may still be usable in non-DHTML browsers. You should always check your DHTML-enabled page in an older browser to see how it looks. Perhaps there isn’t too much you need to do to degrade the DHTML such that the page is acceptable in older browsers.

The ultimate responsibility for deciding your compatibility strategy with older browsers rests with you and your perceptions about your page visitors. If they are in need of vital information from your site and it is readable in non-DHTML browsers, then that may be enough. Otherwise, you must provide a separate content path for both levels of browsers, much as you may be doing for scriptable versus nonscriptable browsers.

A DHTML Application Example

I have created three versions of the same application to demonstrate scripting DHTML in a cross-platform environment (this chapter), in Navigator 4 (Chapter 42), and in Internet Explorer 4 (Chapter 43). The goal is to let those readers who are interested in any one or more of these deployment scenarios see how scripting is done in each. Rather than custom-fit an application that best fits each scenario, I would rather demonstrate the thrills and pitfalls of implementing the same idea under all circumstances.

The application is a jigsaw puzzle game using pieces of a map of the "lower 48" U.S. states (I think everyone would guess where Alaska and Hawaii go on a larger map of North America). I chose this application because it allows me to demonstrate several typical tasks you might want to script in DHTML: hiding and showing elements; handling events for multiple elements; tracking the position of an element with the mouse cursor; absolute positioning of elements; changing the z-order of elements; changing element colors; and animating movement of elements.

As with virtually any programming task, the example code here is not laid out as the quintessential way to accomplish any given task. Each author brings his or her own scripting style, experience, and implementation ideas to a design. Very often, you have available several ways to accomplish the same end. If you find other strategies or tactics for the operations performed in these examples, it means you are gaining a good grasp of both JavaScript and Dynamic HTML.

The puzzle design

Figure 41-1 shows the finished map puzzle with the game in progress. To keep the code to a reasonable length, the example provides positionable state maps for only seven western states. Also, the overall design is intentionally spartan so as to place more emphasis on the PTs and their scripting, rather than on fancy design.

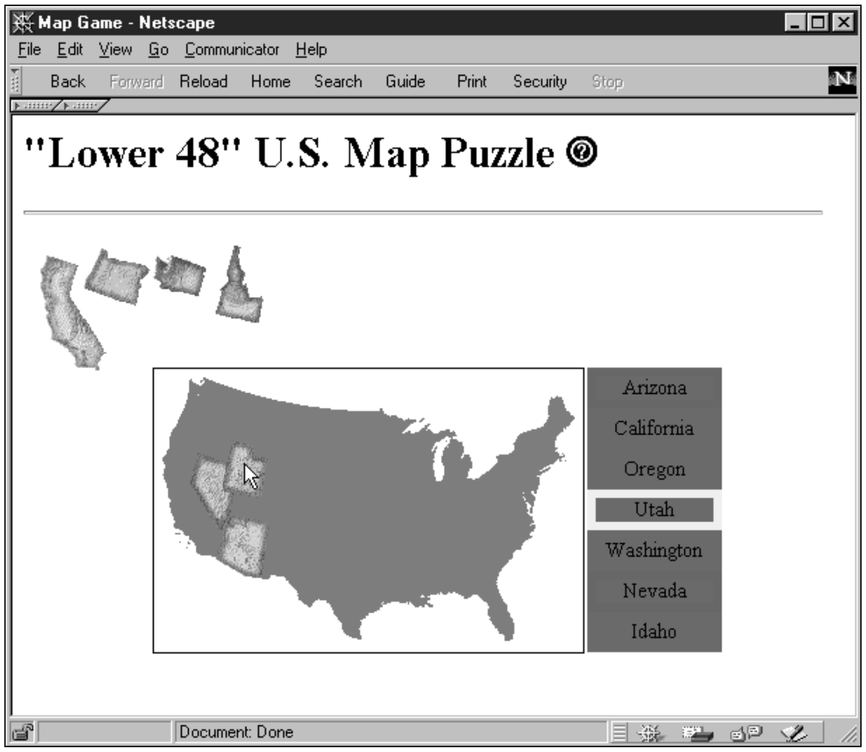
Map Game - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
"Lower 48" U.S. Map Puzzle
Arizona
California
Oregon
Utah
Washington
Nevada
Idaho
Document: Done

Figure 41-1: The puzzle map game DHTML example (Image courtesy Cartesia Software—www.map-art.com)

When the page initially loads, all the state maps are presented across the top of the puzzle area. The state labels all have a red background, and the silhouette of the continental United States has no features in it. To the right of the title is a question mark icon. A click of this icon causes a panel of instructions to glide to the center of the screen from the right edge of the browser window. That panel has a button that hides the panel.

To play the game (there is no scoring or time keeping in this simplified version), a user clicks once on a state map to "pick it up." While a state map is picked up, its label background to the right of the main map turns yellow to highlight the name of the state being worked on. When it is picked up, the state map tracks the position of the mouse cursor. The user then tries to position each state where it belongs on the map, just like a jigsaw puzzle. To release the state in its trial position, the user clicks the mouse again. If the state is within a four-pixel square region around its true location, the state snaps into its correct position and the corresponding label background color turns green. If the map is not dropped close enough to its destination, the label background reverts to red, meaning that the state still needs to be placed.

After the last state map is dropped into its proper place, all the label backgrounds will be green, and a congratulatory message is displayed where the state map pieces originally lay. Should a user then pick up a state and drop it out of position, the congratulatory message disappears.

I had hoped that all versions of the application would look the same on all platforms. They do, with one exception. The cross-platform version shown in this chapter displays one idiosyncrasy on Navigator 4 browsers. I'll have more to say about it later, but suffice it to say that the background color property of the labels is treated differently in Navigator than in Internet Explorer. When implemented as Netscape layers, however, the look is exactly as intended. Therefore, if you try this chapter's version on both Navigator and Internet Explorer, you will see minor differences in the way the labels are colored (red, yellow, and green) during game play. And yet the platform-specific versions in Chapters 42 and 43 look identical to each other.

Implementation details

Due to the number of different scripted properties being changed in this application, I decided to implement a lot of the cross-platform scripting as a custom API loaded from an external .js file library. The library contains functions for most of the scriptable items you can access in DHTML. Having these functions available simplified what would have been more complex functions in the main part of the application. Since the library requires the platform Boolean flags described earlier in this chapter, all functions in the document can use them if needed for some small platform-specific branch detail.

Although I frown on using global variables except where absolutely necessary (like for the platform-specific flags used throughout the application), I needed to assign a few more globals for this application. All of them store information about the state map and state currently picked up by the user. This information needs to survive the invocations of many functions between the time the state is picked up until it is dropped and checked against the "database" of state data.

That database is another global object. Constructed as a multidimensional array, each "record" in the database stores several fields about the state, including its destination coordinates inside the outline map and a Boolean field to store whether the state has been correctly placed in position.

One final point about the overall structure covers the different ways Navigator and Internet Explorer handle events. Rather than define event handlers for each state map object, I wanted the scripts to use a more generic approach. For Navigator that means capturing events at the document level and figuring out which object has been clicked on; for Internet Explorer, it means letting the event object reveal which item has been clicked on. More importantly, I had to make sure that the different ways of handling events didn't step on each other. In some cases, events are handled the same way in both platforms.

The custom API

To begin the analysis of the code, I start with the external .js library file that contains the custom API. Listing 41-1 contains that code. It begins with declaration and initialization of key global variables used throughout the API code. As you can tell from the code and the discussions earlier in this chapter, the API functions use both the platform branching style for some operations and the platform equivalency approach for others. For your convenience, I interlace further commentary amid the long listing.

Listing 41-1: **The Custom API (DHTMLapi.js)**

```
// Global variables
var isNav4, isIE4
var range = ""
var styleObj = ""
if (navigator.appVersion.charAt(0) == "4") {
       if (navigator.appName == "Netscape") {
          isNav4 = true
          insideWindowWidth = window.innerWidth
       } else {
          isIE4 = true
          range = "all."
          styleObj = ".style"
       }
}
```

Many of the functions in this API accept either an object name or object reference as a parameter. The following function processes the first parameter such that object name strings are converted to object references (using the platform equivalency approach), while object references are passed through. This function is called by many functions in the API.

```
// Convert object name string or object reference
// into a valid object reference
function getObject(obj) {
       var theObj
       if (typeof obj == "string") {
```

(continued)

Listing 41-1 *(continued)*

```
            theObj = eval("document." + range + obj + styleObj)
        } else {
            theObj = obj
        }
        return theObj
}
```

The next function, `shiftTo()`, is analogous to Navigator's `layer.moveTo()` method. In fact, for Navigator browsers, that is the method invoked here. But for Internet Explorer, the same action requires adjusting two positional properties, `pixelLeft` and `pixelTop`. Even though the adjustments are made in separate statements, the action on the screen does not follow the action statement-by-statement. Internet Explorer appears to buffer the statements so that the repositioning appears as a single shift.

This `shiftTo()` function accepts either an object name (string) or object reference. As you will see in the main program code, when a user clicks on a state map, the positionable thingy containing the map image becomes the selected object. It is that object that is usually passed to this function.

```
// Positioning an object at a specific pixel coordinate
function shiftTo(obj, x, y) {
        var theObj = getObject(obj)
        if (isNav4) {
            theObj.moveTo(x,y)
        } else {
            theObj.pixelLeft = x
            theObj.pixelTop = y
        }
}
```

The `shiftBy()` function mimics Navigator's `layer.moveBy()` method. The second and third parameters represent the number of pixels that the object should be moved on the page. A positive number means to the right or down; a negative number means to the left or up; a value of zero means no change to the axis. For Navigator, the script uses the `layer.moveBy()` method. But for Internet Explorer, the script uses the add by value operator to change the current value of the style object's `pixelLeft` and `pixelTop` properties.

```
// Moving an object by x and/or y pixels
function shiftBy(obj, deltaX, deltaY) {
        var theObj = getObject(obj)
        if (isNav4) {
            theObj.moveBy(deltaX, deltaY)
        } else {
            theObj.pixelLeft += deltaX
            theObj.pixelTop += deltaY
        }
}
```

Both platforms use the same property name for setting the stacking order of positionable thingies. Therefore, most of the `setZIndex()` function is devoted to handling the string and object versions of the first parameter.

```
// Setting the z-order of an object
function setZIndex(obj, zOrder) {
       var theObj = getObject(obj)
       theObj.zIndex = zOrder
}
```

Each platform has its own way of referring to the background color. The `setBGColor()` function applies the correct syntax for the current platform.

```
// Setting the background color of an object
function setBGColor(obj, color) {
       var theObj = getObject(obj)
       if (isNav4) {
          theObj.bgColor = color
       } else {
          theObj.backgroundColor = color
       }
}
```

I find the allowable values for the `visibility` property very unprogrammatic (I expect a Boolean value). Therefore, to make the process of showing and hiding elements more logical to me, I created API functions called `show()` and `hide()`.

```
// Setting the visibility of an object to visible
function show(obj) {
       var theObj = getObject(obj)
       theObj.visibility = "visible"
}

// Setting the visibility of an object to hidden
function hide(obj) {
       var theObj = getObject(obj)
       theObj.visibility = "hidden"
}
```

Because of the different property names for coordinate locations in each platform, I created two functions that extract the left and top positions for any object. The API takes care of the platform specifics, while code in the main program merely calls for the coordinates.

```
// Retrieving the x coordinate of a positionable object
function getObjectLeft(obj)  {
       var theObj = getObject(obj)
       if (isNav4) {
          return theObj.left
       } else {
          return theObj.pixelLeft
       }
}
```

(continued)

Listing 41-1 *(continued)*

```
// Retrieving the y coordinate of a positionable object
function getObjectTop(obj)  {
       var theObj = getObject(obj)
       if (isNav4) {
          return theObj.top
       } else {
          return theObj.pixelTop
       }
}
```

The above API is generalizable enough to be used as a library with any cross-platform DHTML application. It could even be used with a platform-specific page. It is more efficient, however, to use a browser's native objects, properties, and methods if you know for sure that users will have only one brand of browser.

The main program

Code for the main program is shown in Listing 41-2. It is a comparatively lengthy document, so I will interlace commentary throughout the listing. Before diving into the code, however, allow me to present a preview of the structure of the document. With one exception (the instructions panel), all PTs are defined as CSS-P items. For such a batch generation of items, I usually set the properties for all items within a `<STYLE>` tag set at the very beginning of the HTML page. After that come the scripts for the page. All of this material is inside the `<HEAD>` tag section. I leave the `<BODY>` section to contain the visible content of the page. This is an organization style that works well for me, but you can adopt any style you like, provided various elements that support others on the page are loaded before the dependent items (for example, define a style before assigning its name to the corresponding content tag's `ID` attributes).

Listing 41-2: **The Main Program (mapgame.htm)**

```
<HTML>
<HEAD><TITLE>Map Game</TITLE>
```

Most of the positionable elements have their CSS-P properties established in the `<STYLE>` tag at the top of the document. The `TYPE` attribute points to the cross-platform compatible version. As you will learn in Chapter 42, Navigator offers another type as well but does accept the standard type.

PTs for this application include a background map, a text label for each state, a map for each state, and a congratulatory message. Notice that the names of the label and state map objects begin with a two-letter abbreviation of the state. This comes in handy in the scripts when synchronizing the selected map and its label.

The label objects will be nested inside the background map object. Therefore, the coordinates for the labels are relative to the coordinate system of the background map, not the page. That's why the first label has a `top` property of zero.

```
<STYLE TYPE="text/css">
       #bgmap {position:absolute; left:100; top:180; width:406;}

       #azlabel {position:absolute; left:310; top:0; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #calabel {position:absolute; left:310; top:29; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #orlabel {position:absolute; left:310; top:58; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #utlabel {position:absolute; left:310; top:87; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #walabel {position:absolute; left:310; top:116; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #nvlabel {position:absolute; left:310; top:145; background-
color:red;
              width:100; height:28; border:none; text-align:center;}
       #idlabel {position:absolute; left:310; top:174; background-
color:red;
              width:100; height:28; border:none; text-align:center;}

       #camap {position:absolute; left:20; top:100; width:1;}
       #ormap {position:absolute; left:60; top:100; width:1;}
       #wamap {position:absolute; left:100; top:100; width:1;}
       #idmap {position:absolute; left:140; top:100; width:1;}
       #nvmap {position:absolute; left:180; top:100; width:1;}
       #azmap {position:absolute; left:220; top:100; width:1;}
       #utmap {position:absolute; left:260; top:100; width:1;}

       #congrats {position:absolute; visibility:hidden; left:20;
top:100; width:1;}
</STYLE>
```

The next statement loads the external .js library file that contains the API described in Listing 41-1. I tend to load external library files before listing any other JavaScript code in the page, just in case the main page code relies on global variables or functions in its initializations.

```
<SCRIPT LANGUAGE="JavaScript" SRC="DHTMLapi.js"></SCRIPT>
```

Now comes the main script, which contains all the document-specific functions and global variables. Global variables here are ready to hold information about the selected state object (and associated details), as well as the offset between the position of a click inside a map object and the top-left corner of that map object. You will see that this offset is important to allow the map to track the cursor at the same offset position within the map. And since the tracking is done by repeated calls to a function (triggered by numerous mouse events), these offset values must have global scope.

(continued)

Listing 41-2 *(continued)*

```
<SCRIPT LANGUAGE="JavaScript">
// Global variables
var offsetX = 0
var offsetY = 0
var selectedObj
var selectedState = ""
var selectedStateIndex
```

As the page loads, it executes the following chunk of script to build a database (multidimensional array) of information about each state. The fields for each record are for the two-letter state abbreviation, the full name (not used in this application, but included for use in a future version), the x and y coordinates (within the coordinate system of the background map) for the exact position of the state, and a Boolean flag to be set to true whenever a user correctly places a state.

Getting the data for the x and y coordinates required some legwork. Once I had the pieces of art for each state and the code for dragging them around the screen, I disengaged the part of the script that tested for accuracy. Instead, I added a statement to the code that revealed the x and y position of the dragged item in the status bar (rather than being bothered by alerts). When I carefully positioned a state in its destination, I copied the coordinates from the status bar into the statement that created that state record. Sure, it was tedious, but once I had that info in the database, I could adjust the location of the background map and not have to worry about the destination coordinates, because they were based on the coordinate system inside the background map.

```
// Create 'database' of state information
function state(abbrev, fullName, x, y) {
       this.abbrev = abbrev
       this.fullName = fullName
       this.x = x
       this.y = y
       this.done = false
}
var states = new Array()
states[0] = new state("ca", "California", 7, 54)
states[1] = new state("or", "Oregon", 7, 24)
states[2] = new state("wa", "Washington", 23, 8)
states[3] = new state("id", "Idaho", 48, 17)
states[4] = new state("az", "Arizona", 45, 105)
states[5] = new state("nv", "Nevada", 27, 61)
states[6] = new state("ut", "Utah", 55, 69)
```

The following function, `getSelectedMap()`, is the most complex of the entire application and runs into the most incompatibility between the Navigator layer object and the Internet Explorer style object. The purpose of this function (invoked by the `engage()` function) is to determine which map object has been clicked on by the user and to set key global variables that other functions will use to accomplish the dragging and release of the object.

As happens with several functions in this application, this function defines one parameter (`e`) to accommodate the Navigator event object. Because the parameter is not used in the Internet Explorer 4 branch, no conflict occurs there.

For Navigator processing, the event object plays a key role. The behavior of layer objects and events does not let the event object's `target` property reveal the specific layer being clicked on (unless I had also defined layer events for every map — not a happy prospect). Therefore, I compare the coordinate of the click against the areas occupied by each state map layer. A `for` loop cycles through the state layers in reverse order (so that if the first layout of loose state maps displays maps overlapped with each other, the comparison is done with the map closest to the user's eye). Testing whether a click is within a rectangular region takes a bit of calculation, as you can see from the lengthy conditions in the `if` construction.

Once a coordinate match is found, the `selectedObj` global is set to the layer object occupying that space. Next, two other global values are established: One for the layer containing the label object, and one for the index value in the state "database" for the current state. Notice that the label objects are nested, which means that the Navigator layer reference must traverse the object hierarchy to reach the actual label layer. Also, to bring the selected map to the top of the heap (so it always floats above other states while selected), I set its `zIndex` property to 100 (via one of the API functions).

For Internet Explorer 4 processing, the image inside the positionable thingy is the item that receives the event. Using Internet Explorer's event object, the script extracts that object (via the `srcElement` property). The actual style object is part of the image's parent element, so the `selectedObj` global is set to that object.

The script still needs to set the other global variables for Internet Explorer 4, so it first extracts the state abbreviation from the image object's `parentElement` ID. Then a `for` loop cycles through the database to see if a match occurs for the state abbreviation. If so, the script assembles the reference to the label's style object with the help of that information. Then it saves the index of the record in the database for that state. Finally, the `zIndex` property is set to 100. It is essential to set the `selectedStateLabel` object inside the `for` loop, because that object must be set only when a state map is clicked on.

Every event (of the mouseDown variety, as explained later) on the page is examined by this function. Therefore, if no match is found in the database for the item being clicked on, it must mean that the item is not a map. All the key global variables must be set to null so that no other functions act on those objects.

```
// Set global values related to the selected state map
function getSelectedMap(e) {
       if (isNav4) {
           var clickX = e.pageX
           var clickY = e.pageY
           var testObj
```

(continued)

Listing 41-2 *(continued)*

```
          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)) {
                selectedObj = testObj
                if (selectedObj) {
                       selectedStateLabel =
document.bgmap.document.layers[states[i].abbrev + "label"]
                       selectedStateIndex = i
                       setZIndex(selectedObj, 100)
                       return
                }
             }
          }
       } else {
          var imgObj = window.event.srcElement
          selectedObj = imgObj.parentElement.style
          if (selectedObj) {
             var stateName = imgObj.parentElement.id.substring(0,2)
             for (var i = 0; i < states.length; i++) {
                if (states[i].abbrev == stateName ) {
                       selectedStateLabel = document.all(stateName +
"label").style
                       selectedStateIndex = i
                       setZIndex(selectedObj,100)
                       return
                }
             }
          }
       }
       selectedObj = null
       selectedStateLabel = null
       selectedStateIndex = null
       return
}
```

The `dragIt()` function, compact as it is, provides the main action in the application by keeping a picked-up state object under the cursor as the user moves the mouse. This function is called repeatedly by the mouseMove event, although the actual event handling methodology varies with platform. In Navigator, the event object is passed as a parameter and used to help calculate the position where the map should be in relation to the event coordinates. Internet Explorer coordinates are extracted from its event object, which is accessible as a window property. Both branches call the same API function to adjust the position of the map object.

```
// Position the map where the cursor is
function dragIt(e) {
       if (selectedObj) {
          if (isNav4) {
             shiftTo(selectedObj, (e.pageX - offsetX), (e.pageY -
offsetY))
```

```
          } else {
              shiftTo(selectedObj, (window.event.clientX - offsetX),
(window.event.clientY - offsetY))
          }
      }
}
```

When a user clicks anywhere on the page of this application, the mouseDown event invokes the `toggleEngage()` function. This function acts primarily as a dispatcher to two branches. If `selectedObj` is a value other than null, then it means that the user has previously picked up a state map, and the mouse action is meant to release the map. But if `selectedObj` is null, then it's time to select a map (if one was clicked).

```
function toggleEngage(e) {
      if (selectedObj) {
          release(e)
      } else {
          engage(e)
      }
}
```

The following function contains the code that "picks up" a state map for the user. This only happens if the call to `getSelectedMap()` passes the tests that indicate a state map has been clicked. If that test succeeds, then the offset coordinates of the click relative to the selected object's location are saved in two global variables (`offsetX` and `offsetY`).

Measuring this offset in each platform requires very different syntax due to the different ways each browser refers to its coordinate space. For Navigator, the offset within an object is the difference between the point in the page minus the location of the object. In Navigator, the page coordinates are not affected by scrolling: The top-left of the page is the top-left of the page, no matter where the page is scrolled. At the same time, the object coordinates are measured with respect to the page. But in Internet Explorer, even though you can get the offset coordinate of the event directly from the event object, the value is measured only within the visible area of the document. Therefore, you must also adjust for any possible scrolling of the page.

```
// select a map, set the global offset values, and turn label color to
yellow
function engage(e) {
      getSelectedMap(e)
      if (selectedObj) {
          if (isNav4) {
              offsetX = e.pageX - selectedObj.left
              offsetY = e.pageY - selectedObj.top
          } else {
              offsetX = window.event.offsetX - document.body.scrollLeft
              offsetY = window.event.offsetY - document.body.scrollTop
          }
          setBGColor(selectedStateLabel,"yellow")
      }
}
```

(continued)

Listing 41-2 *(continued)*

When a user drops the currently selected map object, the `release()` function invokes the `onTarget()` function to find out if the current location of the map is within range of the desired destination. If it is in range, the background color of the state label object is set to green, and the `done` property of the selected state database entry is set to true. One additional test looks to see if all the `done` properties are true in the database. If so, the `congrats` object is shown. But if the object is not in the right place, the label reverts to its original red color. In case the user moves a state that was previously okay, its database entry is also adjusted. No matter what the outcome, however, the user has dropped the map, so key global variables are set to null and the layer order for the item is set to zero (bottom of the heap) so it doesn't interfere with the next selected map.

```
// Drop map object and see if it's in range
function release(e) {
        if (onTarget(e)) {
           setBGColor(selectedStateLabel, "green")
           states[selectedStateIndex].done = true
           if (isDone()) {
              show("congrats")
           }
        } else {
           setBGColor(selectedStateLabel, "red")
           states[selectedStateIndex].done = false
           hide("congrats")
        }
        setZIndex(selectedObj, 0)
        selectedObj = null
        selectedState = null
}
```

To find out if a dropped map is in its correct position, the `onTarget()` function first calculates the target spot on the page by adding the location of the bgmap object to the coordinate positions stored in the states database. Since the bgmap object doesn't come into play in other parts of this script, it is convenient to pass merely the object name to the two API functions that get the object's left and top coordinate points.

Next, the script uses platform-specific properties to get the object's current location. The Navigator version is relative to the entire page, while the Internet Explorer version is relative to the visible portion of the page showing in the browser.

A large `if` condition checks whether the object's coordinate point is within a 4-pixel square region around the destination point. If you wanted to make the game easier, you could increase the cushion values from 2 to 3 or 4.

If the map is within the range, the script calls the `shiftTo()` API function to snap the map into the exact destination position, and reports back to the `release()` function the appropriate Boolean value.

```
// See if item is in range of desired location
function onTarget(e) {
       var x = states[selectedStateIndex].x + getObjectLeft("bgmap")
       var y = states[selectedStateIndex].y + getObjectTop("bgmap")
       if (isNav4) {
          var objX = selectedObj.pageX
          var objY = selectedObj.pageY
       } else {
          var objX = selectedObj.pixelLeft
          var objY = selectedObj.pixelTop
       }
       if ((objX >= x-2 && objX <= x+2) && (objY >= y-2 && objY <=
y+2)) {
          shiftTo(selectedObj, x, y)
          return true
       }
       return false
}
```

A simple `for` loop cycles through the states database to see if all of the `done` properties are set to true. When they are, the `release()` function (which calls this `isDone()` function) displays the congratulatory object.

```
// See if all states are "done"
function isDone() {
       for (var i = 0; i < states.length; i++) {
          if (!states[i].done) {
             return false
          }
       }
       return true
}
```

The `help` PT is created differently on this page, as discussed later in the chapter. When the user clicks on the Help button at the top of the page, the instructions panel flies in from the right edge of the window. The `showHelp()` function begins the process by setting its location to the current right window edge, bringing its layer to the very front of the heap, showing the object, and then initiating an interval mechanism that repeatedly calls `moveHelp()`.

```
// Show the help object and start the animation
function showHelp() {
       var objName = "help"
       shiftTo(objName, insideWindowWidth, 80)
       setZIndex(objName,1000)
       show(objName)
       intervalID = setInterval("moveHelp()", 1)
}
```

In the `moveHelp()` function, the `help` object is shifted in 5-pixel increments to the left. The ultimate destination is the spot where the object is in the middle of the browser window. That midpoint must be calculated each time, because the window may have been resized. Unfortunately, Navigator does not provide a mechanism for extracting the width of a layer, so the calculation for the union of the window and layer midpoints is hard-wired to the pixel measure of the layer object.

(continued)

Listing 41-2 *(continued)*

This function is called repeatedly under the control of a `setInterval()` method in `showHelp()`. But when the object reaches the middle of the browser window, the interval ID is canceled, which stops the animation.

The `help` object processes a mouse event to hide the object. An extra `clearInterval()` method is called here in case the user clicks on the object's Close button before the object has reached midwindow (where `moveHelp()` cancels the interval). The script also shifts the position to the right edge of the window, but it isn't absolutely necessary, since the `showHelp()` method positions the window there.

```
// Fly to the left until reaching mid-window
function moveHelp() {
        shiftBy("help",-5,0)
        var objectLeft = getObjectLeft("help")
        if (objectLeft <= (insideWindowWidth/2) - 150) {
            clearInterval(intervalID)
        }
}
// Hide the help window
function hideMe() {
        clearInterval(intervalID)
        hide("help")
        shiftTo("help", insideWindowWidth, 80)
}
```

The document's `onLoad=` event handler calls two functions. The first is `setNSEvents()`, which turns on event capturing for the document and the help layer. Because the help layer is defined so late in the document, it is safest to initialize its events after the page has loaded.

Each object captures different events. The document captures mouseDown and mouseMove events (for picking up, moving, and dropping state maps); the help layer object captures only click events (for hiding the layer).

The second function called at load time is a general-purpose `init()` function. It processes miscellaneous housekeeping items for each platform. Netscape's implementation of CSS1 exhibits odd behavior for setting the background color, as is needed for the labels. While it sets the background color initially, the color does not extend to the edge of the specified height and width of the object. If you then set the layer's `bgColor` property to a color, the gaps between the originally colored space and the edges take on the new color — but the original color stays in the center. To fill out the edge space with the red color at startup time, the `init()` function cycles through all the label objects and sets their background colors to red.

For convenience in moving the help window in Internet Explorer 4, I set a global variable (`insideWindowWidth`) to hold the width of the browser window. I establish this value at load time and also whenever a user resizes a window (as set in the `setWidth()` function that is called by the document's resize event).

```
// Initialize event capturing for Navigator
function setNSEvents() {
        if (isNav4) {
           document.captureEvents(Event.MOUSEDOWN | Event.MOUSEMOVE)
           document.onMouseDown = toggleEngage
           document.onMouseMove = dragIt
           document.help.captureEvents(Event.CLICK)
           document.help.onClick = hideMe
        }
}
// Miscellaneous inits
function init() {
        if (isNav4) {
           for (var i = 0; i< states.length; i++) {

setBGColor((document.bgmap.document.layers[states[i].abbrev +
"label"]),"red")
           }
        } else if (isIE4) {
           insideWindowWidth = document.body.scrollWidth
        }
}
// Reset window width variable upon resize
function setWidth() {
        if (isIE4) {
           insideWindowWidth = document.body.scrollWidth
        }
}
</SCRIPT>
</HEAD>
```

Now comes the part of the document that generates the visible content. Several event handlers are defined in the `<BODY>` tag. Only the `onLoad=` and `onResize=` event handlers are observed by both browsers. Document-level event capture of the mouseDown and mouseMove events for Navigator were handled in the preceding scripts. But Internet Explorer 4 provides these event handlers for the document object, so they can be defined in the traditional event handler format. Navigator simply ignores them.

```
<BODY onLoad="init(); setNSEvents()" onResize="setWidth()"
onMouseDown="toggleEngage()" onMouseMove="dragIt()">
<H1>"Lower 48" U.S. Map Puzzle&nbsp;<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>
```

To be CSS-P friendly, the styles defined early in the document are assigned to block HTML tags. The positionable object for the background map is set up to be a container for all the label objects. Thus, all position coordinates are relative to the coordinate space of the bgmap object.

```
<DIV ID=bgmap><IMG SRC="us11.gif" WIDTH=306 HEIGHT=202
BORDER=1>&nbsp;</IMG>
```

(continued)

Listing 41-2 *(continued)*

```
<DIV ID=azlabel>Arizona</DIV>
<DIV ID=calabel>California</DIV>
<DIV ID=orlabel>Oregon</DIV>
<DIV ID=utlabel>Utah</DIV>
<DIV ID=walabel>Washington</DIV>
<DIV ID=nvlabel>Nevada</DIV>
<DIV ID=idlabel>Idaho</DIV>
</DIV>
```

HTML block tags for the map objects appear later in the document than the background map and labels, even though the state maps are positioned higher in the document. In DHTML, it is not the tag location that determines where objects are placed, but rather the position attributes assigned to the style or layer.

```
<DIV ID=camap><IMG SRC="ca.gif" WIDTH=47 HEIGHT=82 BORDER=0></DIV>
<DIV ID=ormap><IMG SRC="or.gif" WIDTH=57 HEIGHT=45 BORDER=0></DIV>
<DIV ID=wamap><IMG SRC="wa.gif" WIDTH=38 HEIGHT=29 BORDER=0></DIV>
<DIV ID=idmap><IMG SRC="id.gif" WIDTH=34 HEIGHT=55 BORDER=0></DIV>
<DIV ID=azmap><IMG SRC="az.gif" WIDTH=38 HEIGHT=45 BORDER=0></DIV>
<DIV ID=nvmap><IMG SRC="nv.gif" WIDTH=35 HEIGHT=56 BORDER=0></DIV>
<DIV ID=utmap><IMG SRC="ut.gif" WIDTH=33 HEIGHT=41 BORDER=0></DIV>

<DIV ID=congrats><FONT
COLOR="red"><H1>Congratulations!</H1></FONT></DIV>
```

In developing this application, I encountered what surely seems like a bug to me. When using CSS-P to define the instructions panel, Navigator 4 exhibited unwanted behavior after the instruction panel was shown and flown into place under script control. Even after hiding the help object, the page no longer received mouse events, making it impossible to pick up a state map once the instructions appeared. The problem did not surface, however, if the help object was defined in the document with a `<LAYER>` tag.

Therefore, I did what I don't like to do unless absolutely necessary: I created branches in the content that used `document.write()` to create the same content with different HTML syntax depending on the browser. For Internet Explorer, the page creates the same kind of block (with the `<DIV>` tag pair) used elsewhere in the document. Positioning properties are assigned to this block via a `STYLE` attribute in the `<DIV>` tag. You cannot assign a style in the `<STYLE>` tag that is visible to the entire document, because that specification and a like-named `<LAYER>` tag get confused.

For Navigator, the page uses the `<LAYER>` tag and loads the content of the object from a separate HTML file (instrux.htm). One advantage I had with the `<LAYER>` tag was that I could assign an initial horizontal position of the help object with a JavaScript entity. The entity reaches into the `window.innerWidth` property to set the `LEFT` attribute of the layer.

```
<SCRIPT LANGUAGE="JavaScript">
if (isIE4) {
        var output = "<DIV ID='help' onClick='hideMe()'
STYLE='position:absolute; visibility:hidden; top:80; width:300;
border:none; background-color:#98FB98;'>\n"
        output += "<P STYLE='margin-
top:5'><CENTER><B>Instructions</B></CENTER></P>\n"
        output += "<HR COLOR='seagreen'>\n<OL STYLE='margin-right:20'>"
        output += "<LI>Click on a state map to pick it up. The label
color turns yellow.\n"
        output += "<LI>Move the mouse and map into position, and click
the mouse to drop the state map.\n"
        output += "<LI>If you are close to the actual location, the
state snaps into place and the label color turns green.\n"
        output += "</OL>\n<FORM>\n<CENTER><INPUT TYPE='button'
VALUE='Close'>\n</FORM></DIV>"
        document.write(output)
} else if (isNav4) {
        var output = "<LAYER ID='help' TOP=80
LEFT=&{window.innerWidth}; WIDTH=300 VISIBILITY='HIDDEN'
SRC='instrux.htm'></LAYER>"
        document.write(output)
}
</SCRIPT>
</BODY>
</HTML>
```

Lessons learned

Once the external cross-platform API was in place, it helped frame a lot of the other code in the main program. The APIs provided great comfort in that they encouraged me to reference a complex object fully in the main code as a platform-shared value (for example, the `selectedObj` and `selectedState` global variables). At the same time, I could reference top-level thingies (that is, nonnested objects) simply by their names when passing them to API functions.

In many respects, the harder task was defining the CSS-P properties and syntax that both browsers would treat similarly. In the case of the label objects, I couldn't reach complete parity in a cross-platform environment (the labels look different in Navigator), and in the case of the help object, I had to code the HTML separately for each platform. Therefore, when approaching this kind of project, work first with the HTML and CSS-P syntax to build the look that works best for both platforms. Then start connecting the scripted wires. You may have to adjust the CSS-P code if you find odd behavior in one platform or the other with your scripting, but it is still easier to work from a good layout.

But without a doubt the biggest lesson you learn from working on a project like this is how important it still is to test an application on Navigator and Internet Explorer. It would be nearly impossible to design a cross-platform application on one browser and have it run flawlessly on the other the first time. Be prepared to go back and forth between browsers, breaking existing working code along the way until you eventually reach a version that works on both.

✦ ✦ ✦