Chapter 47: Cross-Browser Dynamic HTML Issues

In This Chapter

Introducing Dynamic HTML

The common denominator of DHTML functionality across browsers

Upgrading to modern compatibility techniques

Level 4 browsers and later — NN4+ and IE4+ — were the first browsers to include World Wide Web technologies that gave page authors far more control over the display and interactive behavior of Web page content. Lumped together under the heading of Dynamic HTML (DHTML), these technologies dramatically extended the simple formatting of standard HTML that page authors had used for years. These days, scripters and designers coming to Web development for the first time take DHTML capabilities for granted; they are probably unaware that plain ol' HTML is little more than a specification to format static text and images on a page.

A lot of what the user gets with DHTML had previously been accomplished only via Java applets and plug-ins, such as Shockwave. Not that DHTML eliminates 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; Document Object Model (DOM); and client-side scripting. To this list I also admit recent advances in Extensible Markup Language (XML), which opens the door to author-generated, page-specific HTML extensions that don't rely on standards bodies or browser

support. It will help your authoring skills if you have a little historical perspective on how the Web arrived at DHTML.

For many years, the HTML standard was intended for the rendering of static content — not much more than an electronic version of a printed page. The most interactive part of a page was a form, which included buttons to click and text boxes to fill in. But for anything to change on the page, the content had to be served up again from the host computer.

Client-side scripting, as first implemented through JavaScript in NN2, opened the way for HTML page to not only contain some "smarts," but also control individual pieces of content on the page without fetching a modified page from the server. At first, only form elements were scriptable. Soon thereafter, images could be swapped, although the rectangular space for the image was fixed when the page loaded. More dynamism accrued to pages in NN4 by way of the layer, which acted like a borderless, transparent or opaque window that could contain its own HTML document content and be positioned anywhere on the page, including overlapping content on the main page or other layers. A layer's entire content could be modified without touching the rest of the page or other layers.

But the real breakthrough in dynamism came in IE4, whose rendering engine permitted any element to be modified, inserted, or removed on the fly, while the rest of the page reflowed its content quickly and automatically in response to the change. At the same time, an accepted standard for style sheets (Cascading Style Sheets) opened the way for scripts to modify the look of content already on the page. Text could change colors when a cursor rolled atop it by either adjusting the style sheet property associated with the text or changing the style sheet rule that applies to the text.

Development activity at both Netscape and Microsoft eventually led to a standard for the Document Object Model as a way for scripts to control HTML content directly. Unfortunately, the browser makers frequently implemented first, and then tried to establish their implementations as standards. Sometimes the implementations were not as complete as the standards became, leaving the browsers in states that only partially implement the standards, while paying homage to legacy implementations. Netscape used the occasion of developing an entirely new code base for what became NN6 to try to sever some ties with the past. In many respects that browser represents the state of the standard art as implemented so far. Newest versions of IE, on the other hand, must try to cater to both the legacy implementation and the standards, creating a massive DOM implementation with significant overlap in functionality with different syntaxes. Thus, the result of proprietary explorations and industry standards is a choice of modern browsers that permit a wide range of dynamic activity on content that reaches the browser. Browsers that had started life as sleepy renderers of a tiny HTML vocabulary have grown into powerful front ends for server applications, if not self-contained applications of a sort that execute entirely on the client computer.

Standards for CSS, DOM, and (ECMA) scripting have been well covered earlier in this book. The purpose of this chapter is to demonstrate approaches to accommodating the

sometimes vast differences in specific implementations of these technologies (including browser-specific variations) to produce content that runs on as many DHTML-capable browsers as possible. Most of the problems, as you are well aware from Chapters 15 and 31, are caused by page authors trying to develop for essentially three different document object models: NN4, IE4+, and W3C DOM (as implemented in IE5+ and NN6).

Striving for Compatibility

With as many as three object models to support (you can, of course, elect to support only a subset of browsers if you like) you should look for ways to minimize your pain. If the NN4 object model is in your mix, you will very likely experience moments of sheer torture, as you try to get even the CSS-supported HTML to behave as it does in browsers of the other object models. Thankfully, the NN4 browser's installed base is shrinking, but for some page authors, it can't disappear quickly enough.

Two keys to survival are among the object models: knowing each DOM's limitations and finding common denominators.

In the area of DHTML limitations, NN4 is the clear winner. Compared to the automatic content reflowing of IE4+ and NN6, the NN4 object model is painfully static. For example, dynamically changing the color of a chunk of text in response to a rollover is a difficult task in NN4 requiring the careful positioning of a layer atop main page text; and making any inline modification to content (other than swapping an image of the same size) is completely out of the question. Between the IE4+ and W3C DOMs, the biggest differences fall more along operating system and browser brand lines. Microsoft takes advantage of the integration of the IE browser and the Windows operating system to such an extent that it can provide IE services that work only on Windows versions of IE. IE/Mac users are out of luck (for data binding or text filters, for instance), as are NN6 users.

Looking for areas of commonality — or at least gaining a clear understanding of where the models diverge — can be a tedious, yet personally rewarding pursuit. For example, one of the biggest problems facing designers for all three DOMs is the way scripts must reference elements that are to be moved or hidden (something that all three object models can do). NN4 requires references that take the layer object structure into account; IE4+ has the Microsoft syntax of `document.all`, which provides a reference avenue to any element whose `ID` attribute is set; and the W3C DOM (as implemented in IE5+ and NN6) users a finger-twisting (albeit now industry standard) `document.getElementById()` method to obtain a reference to any ID'd element.

As soon as your script has a valid reference to an element, the next step is to read or write some property, or invoke some method of that object that governs the element's position (and possibly other style) attributes. Here, again, the object models diverge, but not quite as severely. NN4 has a singular implementation that provides properties and methods of positioned elements (`layer` objects) directly; the IE4+ and W3C DOMs, on the other

hand, work their positioning magic through the `style` property of a positioned element. In some cases the "last-dot" property names are identical across all three models (for example, `document.myLayer.zIndex`, `document.all.myLayer.style.zIndex`, and `document.getElementById("myLayer").style.zIndex`). Building a reference to reach that last dot, though, is where some of your hard work must go.

Each DOM also has its own event model. Whereas IE5+ overlaps its DOM features with both the IE4+ and to some extent the W3C DOM, the event models don't follow the same lines of implementation. As of IE5.5/Windows and IE5/Mac, IE does not implement any of the W3C DOM event model, although NN6 does.

The bottom line, then, is letting your scripts decide how to perform actions based on the browser version is not a good idea. Instead, the scripts should be smart enough to act based on the capabilities of the browser that is currently running the script. As you see in the rest of this chapter, it is possible to develop fairly sophisticated DHTML into a page and make it work with all three DOMs without one iota of browser version detection.

Working Around Incompatibilities

To create DHTML for multiple DOMs, you must find ways to accommodate incompatible object references and occasionally incompatible property names. Scripting gives you several alternatives to working your way around these potential problems. Some of the approaches you can take are now passe, but they are described here partly for the sake of historical reference, but also because you will see many instances of these approaches taken in legacy DHTML applications from the days when authors had to worry about only two DOMS (NN4 and IE4). The real "meat" of this discussion comes later, when you learn more about object detection and custom APIs.

Old-fashioned compatibility tricks

In a simpler time (until late 2000), it was possible to write cross-browser DHTML applications that had to run on only two classes of browser: NN4 and IE4. Two approaches to writing code for these two DOMs grew in popularity: inline branching and platform equivalency. They are described here, not for you to apply, but for you to understand what the pioneers did, in case you encounter their code in your Web surfing.

Inline branching

The idea behind inline branching is that your scripts will use `if...else` decisions to execute one branch of code for one browser and another branch for the other browser. 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 (parseInt(navigator.appVersion) == 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. This code, written when the browsers were still at Version 4, was remarkably prescient. My concern at the time was that DHTML was so volatile that it was unknown if future browser versions would be backward compatible with the code to be run inside branches governed by the two global variables. As it turned out, NN6 (whose `navigator.appVersion` reports 5) is not backward compatible with the layer structure of NN4, so that locking the NN4 branches to NN4 became a good thing. On the IE side, the `navigator.appVersion` property continues to report 4, even through IE5.5, which is backward compatible with IE4. Thus, any branch dedicated to IE4 executes under this scheme and remains syntactically accurate.

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 (parseInt(navigator.appVersion) == 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 (and unsupported browsers redirected elsewhere), 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 use the flags as follows:

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

As the browser DOMs evolve, expand, and fragment, inline branching becomes increasingly less practical. With so many permutations of DOM according to browser brand, browser version, and operating system, you can drive yourself crazy trying to accommodate them all and maintain the code going forward. This approach also eliminates from consideration any non-NN or non-IE browser (such as Opera), which may have the capabilities needed to play your DHTML scripts. This approach also limits the possibility that future browsers with higher `navigator.appVersion` values can take advantage of your scripts.

Platform equivalency

Another technique attempts to limit the concern for the different ways each platform refers to a positionable element (because cross-browser DHTML is pretty much limited to the properties affecting positionable elements). If you examine the formats for each platform's object references, you see that all formats contain a reference to the `document` and to the object name or ID. The IE4+ DOM syntax also includes property words, such as `all` and `style`. If you assign these extra property names to variables for IE4 and leave those variables as empty strings for NN4, you can assemble an object reference for those two platforms in one statement.

To begin using this technique, set two global variables that store reference components for the scope (`all` in IE4) and the `style` object (`style` in IE4):

```
var range = ""
var styleObj = ""
if (parseInt(navigator.appVersion) == 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 NN4 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
    }
}
```

As soon as 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"
```

The nested layer situation is not the only potential problem for the platform equivalency approach. In fact, the W3C DOM format for referencing objects (using the `document.getElementById()` method) makes for some hair-raising string assembly and global variable assignment. Another truly negative aspect is the frequent usage of the `eval()` function. As mentioned in Chapter 42, this function is a performance speed thief.

Modern approaches to compatibility

While in-line branching and platform equivalency were suitable for their generations, the profusion of browser versions calls for better approaches to simplifying authoring for multiple DOMs. Techniques more suitable for today — object detection and custom APIs — are not really new. But these techniques are the preferred way to build cross-browser scripts with an eye to compatibility both backward and forward.

Object detection

The subject of object detection has been mentioned in several places in earlier chapters of this book. The technique has been used for a long time to let a browser not equipped to handle image objects gracefully skip over image swapping script segments:

```
if (document.images) {
    // statements to work with image objects
}
```

If there is no `document.images` property for a browser, the condition evaluates to `undefined`, which the condition treats as being `false`.

But object detection has also been misused in the past, especially in the DHTML realm, to substitute for browser version detection. For example, if a browser supported the `document.all` collection, a global variable was set to indicate that the browser was IE4 or later; the existence of `document.layers` supposedly meant that the browser was NN4. While both of those assertions are true (as of the browsers released so far), it was a mistake to link a browser version with the existence of an object or property.

Instead, object detection should be used only if your script statements will be addressing that object, just as the `document.images` condition does in the previous example.

To demonstrate this tactic, consider the need to assemble a reference to an object so that it is ready to have one of its DHTML properties adjusted. Each of the three DOMs has its own syntax for assembling such a reference, and each syntax relies on the existence of a particular object or property. The function shown in Listing 47-1 (not on the CD-ROM by itself, but included in Listing 47-2) lets you pass the name or ID of a positioned element (either in string form or object form) to receive back a valid reference to the object with which style-related properties are associated — all without resorting to the `eval()` function in any form:

Listing 47-1
Using Object Detection to Assemble an Element Object Reference

```
function getObject(obj) {
    var theObj
    if (document.layers) {
        if (typeof obj == "string") {
            // just one layer deep
            return document.layers[obj]
        } else {
            // can be a nested layer
            return obj
        }
    }
    if (document.all) {
        if (typeof obj == "string") {
            return document.all(obj).style
        } else {
            return obj.style
        }
    }
    if (document.getElementById) {
        if (typeof obj == "string") {
            return document.getElementById(obj).style
        } else {
            return obj.style
        }
    }
    return null
}
```

The primary object detection for each of the three sections of this function looks for the presence of categories of objects (`document.layers` and `document.all`) or a particular method (`document.getElementById()`), and then — this is the important part — the script uses those detected objects in the statements. The script doesn't know IE4 from NN6; it *does* know how to derive valid references for three different object models, and employs the syntax of the first one for which the associated object property or method is supported.

In practice, the order of the three sections should have no bearing on your scripts, but you should be aware of one subtlety: IE5+ can work with either of the last two sections, because those browsers detect `document.all` and `document.getElementById` as valid references. If you were to switch the position of the last two sections, then IE5+

would be using W3C DOM terminology. The results, however, are the same: A valid reference to the `style` object associated with an element.

Custom APIs

Notions of object detection and simplifications of your scripts come together in the final approach to building cross-browser DHTML: Writing a custom API (Application Programming Interface). A JavaScript custom API is a library of functions you design to act 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-browser DHTML page, an API can offer a single function that smoothes over the differences in object references and/or property names among several platforms. Your custom function provides a single access point that is consistent across all platforms. In essence, you are creating your own metavocabulary for methods and property settings.

The element object reference maker in Listing 47-1 is a good start for such an API, because all other functions for moving, hiding, showing, and changing the stacking order of a positionable element need a valid style-oriented reference to the element. Now look at a function from an API whose job is to alter the stacking order of a positionable element:

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

Your main page script would use the ID of the positioned DIV element as the first parameter to this function, with an integer indicating the value that would be assigned to the element's style sheet `z-Index` attribute:

```
setZIndex("myLayer", 100)
```

All of the branching for the various DOMs in this function is done in the `getObject()` function (Listing 47-1), which returns the valid reference for whichever of the three supported DOMs is running the script. All three DOMs, it turns out, have the same `zIndex` property representing the `z-Index` style attribute, so that no further branching is needed here.

As one more example, the next API function offers an interface to incompatible ways of adjusting the location of a positionable element. In this case, the act of moving an element has different syntax in different DOMs. One group (NN4 for layers) uses the `moveTo()` method; the rest support `left` and `top` properties of their `style` object:

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y) {
    var theObj = getObject(obj)
    if (theObj.moveTo) {
        theObj.moveTo(x,y)
    } else if (typeof theObj.left != "undefined") {
        theObj.left = x
        theObj.top = y
    }
}
```

Notice one workaround, which, on the surface, isn't pretty: The second branch must perform an odd way of object detection. We're stuck with having to make a tradeoff when it comes to checking for the existence of a style property. If the page uses style sheets defined in `<STYLE>` tags (or imported into the page from external style sheet files), the element affected by the rule does not yield the rule's property values through the element's `style` property. The property exists, but its value in this case (or until it is set by script) is an empty string. IE5 provides a `currentStyle` property to give us the effective values, but that property is not (yet) a part of the DOM standard. But even if you assign the style sheet via the element's `STYLE` attribute (in which case the style property values come through), detecting the presence of the property with the conditional expression

```
if (theObj.left)
```

is not practical here anyway. If the effective value of the `left` and `top` properties were an empty string (or zero for a numeric style property value), the conditional expression would evaluate to the equivalent of `false`, making it appear as though the property doesn't exist. To validate the existence of the property, the conditional expression verifies that the value of a named property has a type other than "undefined." It may seem like a long way to go to prove the existence of a property, but it works, even if the value is an empty string or zero.

It is important that both branches perform object detection. Although it is unlikely (but, as we learned from the transition between NN4 and NN6, not impossible), if a future browser should completely alter its vocabulary, omitting the objects being detected here, the function ends gracefully, without generating script errors.

An API is usually best deployed as an external `.js` file. One such API file is described later in this chapter. Bear in mind, however, that a lengthy API gets downloaded to the browser, no matter how much or how little of it your main scripts use. Blindly linking in a big library just to use a few of its functions is a mistake. You serve your users better if you create a subset of the API, and link the subset to the page (or drop the few functions directly into the page's scripts if the combination is not reused on a lot of pages).

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 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 (center or right) is at work. This goes for elements that you design to be DHTML-positioned to sit offscreen (perhaps with a clickable tab) until called by the user. An element defined as being hidden or not

displayed 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 so 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 that information 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 API Example

Now it's time to get to a real DHTML API that you can use and build upon for your own applications. Listing 47-2 contains the API code, which is most likely to be deployed as an external `.js` library file. In fact, this API is used as-is in a map puzzle game application in Chapter 56. You can see there how it is used to control element positioning, dragging, and layering for the three DOM families. The code in Listing 47-2 is longer than most listings in this book, so for your convenience, I interlace commentary amid the long listing.

No global variables are needed for this API. Because all browser branching is performed via object detection, there is no need for browser version detection. Instead, the library starts with the `getObject()` function shown earlier in this chapter. Virtually every other function in this library makes a trip to `getObject()` to convert the name of the object passed as a parameter to an object reference whose positionable (or other style-related property) can be adjusted.

Listing 47-2
The Custom API (DHTMLapi.js)

```
// convert object name string or object reference
// into a valid object reference ready for style change
function getObject(obj) {
    var theObj
    if (document.layers) {
        if (typeof obj == "string") {
            return document.layers[obj]
        } else {
            return obj
        }
    }
    if (document.all) {
        if (typeof obj == "string") {
            return document.all(obj).style
        } else {
            return obj.style
```

```
        }
    }
    if (document.getElementById) {
        if (typeof obj == "string") {
            return document.getElementById(obj).style
        } else {
            return obj.style
        }
    }
    return null
}
```

A pair of functions handles all motion of positionable elements. The first function, `shiftTo()` takes three parameters: the ID of the object being moved, and the horizontal and vertical pixel coordinates of the top-left corner of the element. The assumption is that the main page script that invokes this library function performs the calculation of the coordinates. You see that code in Chapter 56. Branches inside this function handle the NN4 `layer.moveTo()` method or the setting of `style` properties for other DOMs. In these other browsers, moving the element requires adjusting two positional properties, left and top. Even though the adjustments are made in separate statements, the action on the screen does not follow the action statement-by-statement. Between screen buffering and quick execution, the repositioning appears as a single shift.

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y) {
    var theObj = getObject(obj)
    if (theObj.moveTo) {
        theObj.moveTo(x,y)
    } else if (typeof theObj.left != "undefined") {
        theObj.left = x
        theObj.top = y
    }
}
```

The `shiftBy()` function mimics NN4'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 NN4, the script uses the `layer.moveBy()` method. But for the rest, the passed values are added to the `left` and `top` properties. Notice that because these properties return strings that include the units for the measurements, the incremental values are added to integer extractions from the current settings. And because the units being used here are the default (pixels), no units have to be assigned with the new values (although they could without penalty).

```
// move an object by x and/or y pixels
function shiftBy(obj, deltaX, deltaY) {
    var theObj = getObject(obj)
    if (theObj.moveBy) {
        theObj.moveBy(deltaX, deltaY)
    } else if (typeof theObj.left != "undefined") {
        theObj.left = parseInt(theObj.left) + deltaX
        theObj.top = parseInt(theObj.top) + deltaY
    }
}
```

Both platforms use the same property name for setting the stacking order of positionable thingies. Therefore, the `setZIndex()` function does little more than convert the object reference and assign the incoming value to the `zIndex` property.

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

NN4 and browsers with `style` objects have their own way of referring to the background color. The `setBGColor()` function applies the correct syntax based on the whichever property is detected in the object.

```
// set the background color of an object
function setBGColor(obj, color) {
    var theObj = getObject(obj)
    if (theObj.bgColor) {
        theObj.bgColor = color
    } else if (typeof theObj.backgroundColor!= "undefined") {
        theObj.backgroundColor = color
    }
}
```

Allowable values for the `visibility` property are very unprogrammatic in my opinion. I expect a Boolean value rather than strings. To accede to reality while making the process of showing and hiding elements more logical to me, I created API functions called `show()` and `hide()`.

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

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

Although the `left` and `top` properties of NN4 layers do not include unit values, it is still safe to use `parseInt()` on the values returned from the properties, whether they be retrieved in NN4 or browsers that have `style` objects (whose properties return units). The need for these API functions came from the way the map puzzle application in Chapter 56 works. For a couple of operations, it calculates the destination for an object with respect to the position of one of the other positioned elements. These functions return the values needed for the main program's calculation. This is also an example of how you may need to embellish the API for your own application.

```
// retrieve the x coordinate of a positionable object
function getObjectLeft(obj)  {
    var theObj = getObject(obj)
    return parseInt(theObj.left)
}

// retrieve the y coordinate of a positionable object
function getObjectTop(obj)  {
    var theObj = getObject(obj)
    return parseInt(theObj.top)
}
```

The previous API is generalizable enough to be used as a library with any cross-platform DHTML application using positioning. The API can 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.