Chapter 52: Application: Outline-Style Table of Contents

In This Chapter

Multiple frames

Clickable images

Custom objects

Image caching

Persistent data

Dynamic HTML positioning

In your Web surfings, you may have encountered sites that implement an expandable, outline type of table of contents. I’ve long thought that these elements were great ideas, especially for sites with lots of information. An outline, such as the Windows Explorer or text-style Macintosh Finder windows, enables the author to present a large table of contents in a way that doesn’t necessarily take up a ton of page space or bandwidth. From listings of top-level entries, a user can drill down to reveal only those items of interest.

No matter how much I like the idea, however, I dislike visiting these sites. A CGI program on the server responds to each click, chews on my selection, and then sends back a completely new screen, showing my choice expanded or collapsed. After working with outlines in the operating system and outliner programs on personal computers, the delays in this processing seem interminable. It occurred to me that implementing the outline interface as a client-side JavaScript can significantly reduce the delay problem and make outlines a more viable interface to a site’s table of contents. This chapter documents the process that went into an early version of the outliner, which works with most older browsers. Some newer versions are also presented.

Design Challenges

The more I looked into implementing an outline in the early scripting days, the more challenges I found ahead of me.

The first problem was making the little icons (widgets) clickable so that they respond to user mouse actions. Even though images are objects in NN3 and IE4+, NN images don’t have mouse-oriented event handlers until you reach NN6 (although you can make some mouse events work in some versions of NN4/Windows). Therefore, it was necessary to surround each image with a link object whose HREF attribute called a `javascript:` URL and function to do the job. This technique also helped solve the next problem.

After a user clicks an outline widget, the script must update the window or frame containing the outline to expand or collapse a portion of the outline. The original design predated dynamically updated pages of IE4 and NN6, so the entire page had to be rewritten. But to make that work, the script needed a way to represent and temporarily preserve the current state of the outline — a line-by-line rundown on whether a line was currently expanded or collapsed. If the script could save that state somewhere, the widget's link HREF attribute could summon a JavaScript function whose job is to perform a soft reload of the current page without reopening it — with the `history.go()` method. Therefore, as a user clicked a widget, the state of the outline created by that click would be generated in the script, saved, and then used to specify the expanded or collapsed state of each line as the page reloaded.

Just when I was congratulating myself on how clever I was, I realized that any attempt to save the state of the new outline in a variable was doomed: Even a soft reload restores variables to their original state. I'd have to find another way to maintain the data.

The first method I used was to store the outline state (a string of `0`s and `1`s, in which a `1` indicated that the item was expanded) in a text box. Text and TEXTAREA objects maintain their contents even through a document reload (but not a reopen). Although this method was convenient, it was ugly because it meant that the field would have to be in the frame. One tactic was to make the frame a non-scrolling frame and stuff the field out of sight by pushing it to the far right with padding spaces inside a `<PRE>`…`</PRE>` tag.

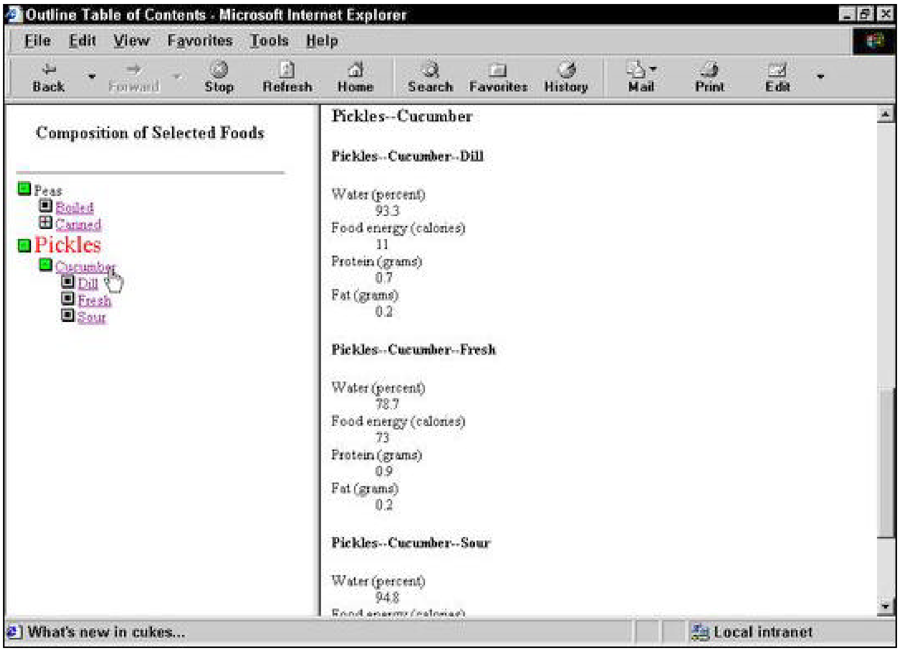
Next, it was time to try Netscape's mechanism for storing persistent data on the client computer: the `document.cookie` property. Cookies are not unique to JavaScript. Any CGI can also store data, such as a user's login name and password for a site, in a cookie. The cookie did the trick. Information about the outline lasts in the cookie of any user's computer only as long as the browser stays running.

Another detail that I wanted to overcome was the initial delay experienced the first time a user clicked one of the collapsed widgets in the outline. At that point, only one of three icon image files had been loaded and cached in the browser. In the very first version of this application for NN2, I arranged to display all three widgets as decoration on the page to get them loaded up front. But with NN3+ and IE4+, I can precache all the widget art files and deploy them instantly when needed.

The Implementation Plan

I admit to approaching the outline technique the first time without a specific data-display goal in mind — not always the best way to go about it. In search of some logical and public domain data that I could use as an example, I came upon the tables of information about food composition (grams of protein, fat, calories, and so on) published by the U.S. government. For this demonstration, I created one HTML document containing data for two hierarchical categories of foods: peas and pickles. At the beginning of each food category, I assigned an anchor to which the text entries of the outline point.

My design for this implementation calls for two frames set up as columns (see Figure 52-1). The narrower left column houses the outline interface. After the frameset loads, the wider right frame initially shows an introductory HTML document. Clicking any of the links in the outline changes the view of the right-hand frame from the introductory document to the food data document. A link at the bottom of the food data document enables the user to view the introductory document again in the same frame, if desired.



Composition of Selected Foods

Peas
Boiled
Canned
Pickles
Cucumber
Dill
Fresh
Sour

Pickles--Cucumber

Pickles--Cucumber--Dill

Water (percent)
93.3
Food energy (calories)
11
Protein (grams)
0.7
Fat (grams)
0.2

Pickles--Cucumber--Fresh

Water (percent)
78.7
Food energy (calories)
73
Protein (grams)
0.9
Fat (grams)
0.2

Pickles--Cucumber--Sour

Water (percent)
94.8
Food energy (calories)

Figure 52-1
The outline in the left frame is dynamic and local

In addition to image caching, NN3 and IE4 gave me reason to make some other improvements to the outliner over a version originally created for NN2. They include

* Adjustable indentation spacing
* Easier specification of widget art files
* Easier way to specify a target frame for the results

* Additional array field for statusbar display text

All adapter-adjustable elements appear near the top of the script to make it easy for scripters without a lot of experience to modify the application for their own sites.

For this fourth edition of the book, a couple of minor improvements make the outliner easier to modify and deploy. First, the tedious sequential numbering of items is gone. Second, performance in NN4 is greatly enhanced with the help of streamlined cookie handling.

The Code

All files for this implementation of the outline are on the CD-ROM accompanying this book, so I display here only the code for the framesetting document (`index.htm`) and the outline (`toc5.htm`). Earlier numbered filenames were used for previous editions of this book.

Setting the frames

To establish the frames, the script creates a two-column format, assigning 35 percent of the page as a column to contain the outline:

```
<HTML>
<HEAD>
<TITLE>Outline Table of Contents</TITLE>
</HEAD>

<FRAMESET COLS="35%,65%">

<NOFRAMES>
<H1>It's really cool...</H1>
<H2>...but only if you a frames-capable browser</H2>
<HR>
<A HREF="index.html">Back </A>
</NOFRAMES>

    <FRAME NAME="Frame1" SRC="toc4.htm">
    <FRAME NAME="Frame2" SRC="intro.htm">
</FRAMESET>
</HTML>
```

Because pages designed for multiple frames and JavaScript don’t fare well in browsers incapable of displaying frames, a good approach is to surround HTML with a `<NOFRAMES>` tag for display to users of old browsers. You can substitute any link you like for the one shown here, which goes back to the main JavaScript page at my Web site.

The names that I assign to the two frames aren’t very original or clever, but they help me remember which frame is which. Because the nature of the contents of the second frame changes (either the introductory document or the data document), I couldn’t think of a good name to reflect its purpose.

Outline code

Now we come to some lengthy code for the outline (in file `toc5.htm`). Much of the code deals with managing the binary representation of the current state of the outline. For each line of the completely exploded outline, the code designates a `0` for a line that has no nested items showing and a `1` for a line that has a nested item showing. This sequence of `0`s and `1`s (as one string) is the road map that the script follows when redrawing the outline. Cues from the `0` and `1` settings let the script know whether it should display a nested item (if one exists) or leave that item collapsed.

To help me visualize the inner workings of these scripts, I developed a convention that calls any item with nested items beneath it a *mother*. Any nested item is that mother's *daughter*. A daughter can also be a mother if it has an item nested beneath it. You see how this plays out in the code shortly.

The food outline document starts out simply enough, with the standard opening of a JavaScript script. The first specification set apart for easy modification is the size of the indentation level in pixels.

```
<HTML>
<HEAD>
<TITLE>Food Selection Outline</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding
// ** BEGIN OUTLINE AUTHOR-ADJUSTABLE SPECIFICATIONS **//

// size of horizontal indent per level
var indentPixels = 20
```

Outline level indentations are controlled by the width of a transparent image file. Indentation size is uniform throughout the outline, and the value is controlled via the `indentPixels` global variable. The image file is actually only a single pixel large, but by setting the width as needed (see the following example), it occupies a known amount of space, without affecting the font characteristics of the outline text.

Two more groups of adjustable items come next. The first group takes care of the widget images. This group is where you specify the filenames for the three widgets and provide the script with their height and width measurements:

```
// art files and sizes for three widget styles
// (all three widgets must have same height/width)
var collapsedWidget = "plus.gif"
var expandedWidget = "minus.gif"
var endpointWidget = "end.gif"
var widgetWidth = 12
var widgetHeight = 12

// Target for documents loaded when user clicks on a link.
// Specify your target frame name here.
var displayTarget = "Frame2"
```

When you design your widget art (if you don't like mine), be sure to design all three to be the same size. This practice prevents scaling of the images later.

If you deploy the outliner for your site, be sure to change the name of the frame assigned to the `displayTarget` global variable. This value eventually becomes part of the text links in the outline. If you want a click of a link to completely replace the current frameset with a different page, then specify `_top` as the display target.

Assembling outline content

The last of the easily modifiable areas defines the content of the outline. After defining the primary array (`db`), a second dimension is added to create an array of custom objects. The `dbRecord` array (defined in the following listing) helps populate the `db` array with specifics provided in the comma-delimited statements here:

```
// Create array object containing outline content and attributes.
// To adapt outline for your use, modify this table.
// Start the array with [1], and continue without gaps to your last item.
// The order of the five parameters:
//    1. Boolean (true or false) whether _next_ item is indented.
//    2. String to display in outline entry (including <FONT> tags).
//    3. URL of link for outline entry; Use empty string ("") for no link
//    4. Integer of indentation level (0 is leftmost margin level)
//    5. String for status line onMouseOver (apostrophes require \\')
var db = new Array()
db[db.length] = new dbRecord(true,"Peas","",0,"")
db[db.length] = new dbRecord(false,"Boiled","foods.htm#boiled",
    1,"Mmm, boiled peas...")
db[db.length] = new dbRecord(true,"Canned","foods.htm#canned",
    1,"Check out canned peas...")
db[db.length] = new dbRecord(false,"Alaska","foods.htm#alaska",
    2,"Alaska\\'s finest...")
db[db.length] = new dbRecord(false,"Low-Sodium","foods.htm#losodium",
    2,"A healthy treat...")
db[db.length] = new dbRecord(true,"<FONT COLOR=red
SIZE=+2>Pickles</FONT>","",0,"")
db[db.length] = new dbRecord(true,"Cucumber","foods.htm#cucumber",
    1,"What\\'s new in cukes...")
db[db.length] = new dbRecord(false,"Dill","foods.htm#dill",
    2,"Pucker up...")
db[db.length] = new dbRecord(false,"Fresh","foods.htm#fresh",
    2,"You\\'d prefer stale?")
db[db.length] = new dbRecord(false,"Sour","foods.htm#sour",
    2,"For sweeties...")
// add more records to complete your outline
// ** END AUTHOR-ADJUSTABLE SPECIFICATIONS **//
```

Each record consists of five items. The first item is a Boolean value, which denotes whether the item is a mother item (that is, the next item in the list is nested one level deeper). The HTML that displays in the outline comes next. This text can be multiple-word strings, or any HTML that you like (some users have assigned `<IMG>` tags to show pictures instead of text). For the third item, you can insert any valid URL, whether it be to a separate site, an anchor in another document (as shown here), or even a `javascript:` URL to execute another function. If you don’t want an entry in the outline to be a link — just plain, flat text — leave this third item as an empty string, as I do here for the topmost items in both categories. The fourth item is a number representing how many nested levels the item has. And finally, the last item is a string containing the text that appears in the statusbar when the user rolls the mouse over the item in the outline. Because of a quirk in the way the statusbar responds to quoted characters, any string literal character (normally preceded with a backslash) requires two backslashes (one to alert the browser of the other).

Be sure to keep the items for the `db` array in the same top-to-bottom order as you'd expect to see in a fully expanded outline. Notice that the index values of the array are automatically inserted for you: The `length` property of an array is always one more than the highest index. By inserting references to the `db.length` property in the brackets, you instruct JavaScript to "walk the ladder" upward from zero. If you move things around the outline, however, don't forget to adjust the indentation levels if they are affected by the content changes.

The bottom of the array creation section marks the end of the code that you need to modify after you deploy the outliner. The rest of the JavaScript code works silently for you, but if you intend to perform further customizations to the outliner, understanding how it all works will help.

On to the constructor function for the `dbRecord` entries: This function is the classic JavaScript way to build a custom object (see Chapter 41):

```
// object constructor for each outline entry
// (see object-building calls, below)
function dbRecord(mother,display,URL,indent,statusMsg){
    this.mother = mother   // is this item a parent?
    this.display = display // text to display
    this.URL = URL         // link tied to text; no link for empty string
    this.indent = indent   // how deeply nested?
    this.statusMsg = statusMsg  // descriptive text for status bar
    return this
}
```

To preload all the images into the browser's image cache, you create new `Image` objects for each and assign the filenames to their `src` properties. Notice that these statements are not in functions, but rather execute as the page loads:

```
// pre-load all images into cache
var fillerImg = new Image(1,1)
fillerImg.src = "filler.gif"
var collapsedImg = new Image(widgetWidth,widgetHeight)
collapsedImg.src = collapsedWidget
var expandedImg = new Image(widgetWidth,widgetHeight)
expandedImg.src = expandedWidget
var endpointImg = new Image(widgetWidth,widgetHeight)
endpointImg.src = endpointWidget
```

Cookie storage

To preserve the binary digit string between redraws of the outline, this script must save the string to a place that won't be overwritten or emptied during the document reload. The `document.cookie` fills that requirement nicely. Excerpting and adapting parts of Bill Dortch's cookie functions (see Chapter 18), this script simplifies the writing of a cookie that disappears when the user quits the browser:

```
// ** functions that get and set persistent cookie data **
// set cookie data
var mycookie = document.cookie
function setCurrState(setting) {
        mycookie = document.cookie = "currState=" + escape(setting)
}
// retrieve cookie data
function getCurrState() {
        var label = "currState="
        var labelLen = label.length
        var cLen = mycookie.length
        var i = 0
        while (i < cLen) {
                var j = i + labelLen
                if (mycookie.substring(i,j) == label) {
                        var cEnd = mycookie.indexOf(";",j)
                        if (cEnd ==     -1) {
                                cEnd = mycookie.length
                        }
                        return unescape(mycookie.substring(j,cEnd))
                }
                i++
        }
        return ""
}
```

A global variable is used to act as a speedy intermediary between the actual browser cookie and the functions here that need to access cookie data. The `setCurrState()` function contains a construction that you don't see much in this book, but is quite valid. Notice the three-piece assignment statement. Evaluation of this statement works from right to left. The rightmost expression concatenates a cookie label and the value passed in as a parameter to the function. Note, too, that the value is passed through the `escape()` function to properly URL-encode the data for the sake of data integrity (so that spaces and odd punctuation don't mess up the mechanism). The concatenated value is assigned to the `document.cookie` property. With the value safely dropped into the cookie (it may be just one of several name/value pairs for this domain), the value of the `document.cookie` property (which includes all name/value pairs for the domain) is assigned to the `mycookie` global variable.

Retrieving information from the cookie still requires a bit of parsing to be on the safe side. If other cookie writing were to come from the current server path, more than one cookie would be available to the current document. Parsing the entire cookie for just the portion that corresponds to the `currState` labeled cookie ensures that the script gets only the data previously saved to that label. In an earlier version of this code, the frequent access to the `document.cookie` property inside the `while` loop of `getCurrState()` wasn't a problem until the sluggish cookie reading performance of NN4 got in the way. Adapting the code to use the global variable for the repetitive parsing of the cookie value rescued the day.

The focal point

The `toggle()` function, which is pivotal in this outline scheme, receives as a parameter the index number of the `db` array element whose content the user just clicked. The purpose of this function is to grab a copy of the current outline state from the cookie, alter the binary representation of the clicked item, and feed the revised binary number back to the cookie (where it governs the display of the outline after the document reloads):

```
// **function that updates persistent storage of state**
// toggles an outline mother entry, storing new value in the cookie
function toggle(n) {
    var newString = ""
    var currState = getCurrState()
    var expanded = currState.charAt(n) // of clicked item
    newString += currState.substring(0,n)
    newString += expanded ^ 1 // Bitwise XOR clicked item
    newString += currState.substring(n+1,currState.length)
    setCurrState(newString) // write new state back to cookie
 }
```

To make this happen, you must extract two pieces of information before any processing: the current state from the cookie and the current setting of the clicked item. The latter is saved in a local variable named `expanded` because its `0` or `1` value represents the expanded state of that particular entry in the outline.

With those information morsels in hand, the script starts building the new binary string that gets written back to the cookie. The new string consists of three pieces: the front part of the existing string up to (but not including) the digit representing the clicked item, the changed entry, and the rest of the original string.

Changing the setting of the clicked item from a `0` to a `1`, or vice versa, is necessary. Although I can implement this task a few different ways (for example, using a conditional expression or an `if…else` construction), I thought I'd exercise an operator that otherwise gets little use: the bitwise XOR operator (`^`). Because the values involved here are `0` and `1`, performing an XOR operation with the value of `1` inverts the original value:

```
0 ^ 1 = 1
1 ^ 1 = 0
```

Okay, perhaps using an XOR operator is showing off. But the experience forced me to understand a JavaScript power that may come in handy for the future.

Selecting a widget image for an entry

At this point, the script starts defining functions to help the script statements in the Body write the HTML for the new version of the outline. The `getGIF()` function determines which of the three widget image files needs to be specified for a particular entry in the outline. The function receives the index value to the `db` array of entries created earlier in the script. As the Body script assembles the HTML for the outline, it calls this function once for each item in the outline. In return, the function provides a reference to one of three `Image` objects created earlier:

```
// **functions used in assembling updated outline**
// returns the proper GIF file name for each entry's control
function getGIF(n, currState) {
    var mom = db[n].mother  // is entry a parent?
    var expanded = currState.charAt(n) // of clicked item
    if (!mom) {
        return endpointWidget
    } else {
        if (expanded == 1) {
            return expandedWidget
        }
    }
    return collapsedWidget
}
```

The decision process for this function first tries to eliminate any item that ends a mother–daughter chain. Any item that is as deeply nested as it can be (which means the item is not a mother) automatically gets the `endpointWidget` image.

Now you're left with trying to figure out whether the item in the display should get an expanded or collapsed icon. The holder of this information is the cookie. Thus, the script extracts the binary setting for the entry under scrutiny. If the cookie shows that entry to be a `1`, it means that the item has nested items showing and that it should get the `expandedWidget` image; otherwise, it should get the `collapsedWidget` image. Notice that you're returning *references* to the `Image` objects, not the names of the image files.

A similar excursion through each item determines what status message is assigned to the `onMouseOver` event handler for each of the widget images. The decision tree is identical to that of the `getGIF()` function:

```
// returns the proper status line text based on the icon style
function getGIFStatus(n, currState) {
    var mom = db[n].mother  // is entry a parent
    var expanded = currState.charAt(n) // of rolled item
    if (!mom) {
        return "No further items"
    } else {
        if (expanded == 1) {
            return "Click to collapse nested items"
        }
    }
    return "Click to expand nested items"
}
```

Initialize the cookie

The final task of the script running in the head is to initialize the cookie if it's empty. Using the length of the `db` array as a counter, you build a string of `0`s, with one `0` for each item in the outline:

```
// initialize 'current state' storage field
if (getCurrState() == "" || getCurrState().length != (db.length)) {
    initState = ""
    for (i = 0; i < db.length; i++) {
        initState += "0"
    }
    setCurrState(initState)
}

// end -->
</SCRIPT>
</HEAD>
```

Each of those 0s in the parameter to the `setCurrState()` function corresponds to a collapsed setting for an entry in the outline. In other words, the first time the outline appears, all items are in the collapsed mode. If you modify the outline for your own use by creating your own `db` array of data, the initial state of the cookie will be set for you automatically based on the length of the `db` array.

Writing the outline

At last we reach the document Body, where the outline is assembled and written to the page. Script statements here are immediate, meaning that they execute while the page loads. I have you begin by initializing some variables that you will need in a moment. The most important variable is `newOutline`, which will be used to accumulate the contents of the outline for eventual writing to the page:

```
<BODY>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
// build new outline based on the values of the cookie
// and data points in the outline data array.
// This fires each time the user clicks on a control,
// because the HREF for each one reloads the current document.
var newOutline = ""
var prevIndentDisplayed = 0
var showMyDaughter = 0
var currState = getCurrState() // get whole state string
document.write("<CENTER><H3>Composition of Selected Foods</H3>")
```

The following section is the real beef of this script: the part that assembles the HTML for the outline that displays as the document loads. In other words, this part must read the current state data from the cookie and assemble widget images and text links according to the map of expanded and collapsed items in the cookie data. These activities take place within a `for` loop that cycles through every item in the database. Each value of the `i` index refers to one listing in the `db` array. Trace the logic of one entry:

```
// cycle through each entry in the outline array
for (var i = 0; i < db.length; i++) {
    var theGIF = getGIF(i, currState)  // get the image
    var theGIFStatus = getGIFStatus(i, currState) // get the status message
    var currIndent = db[i].indent      // get the indent level
    var expanded = currState.charAt(i) // current state

    // display entry only if it meets one of three criteria
    if (currIndent == 0 || currIndent <= prevIndentDisplayed ||
        (showMyDaughter == 1 &&
        (currIndent - prevIndentDisplayed == 1))) {
        newOutline += "<IMG SRC=\"filler.gif\" HEIGHT = 1 WIDTH =" +
            (indentPixels * currIndent) + ">"
        newOutline += "<A HREF=\"javascript:history.go(0)\" " +
            "onMouseOver=\"window.status=\'" + theGIFStatus +
            "\';return true;\" onClick=\"toggle(" + i + ");return " +
            (theGIF != endpointWidget) + "\">"
        newOutline += "<IMG SRC=\"" + theGIF + "\" HEIGHT=" +
            widgetHeight + " WIDTH=" + widgetWidth + " BORDER=0></A>"
        if (db[i].URL == "" || db[i].URL == null) {
            newOutline += " " + db[i].display + "<BR>"    // no link
        } else {
            newOutline += " <A HREF=\"" + db[i].URL + "\" TARGET=\"" +
                displayTarget + "\" onMouseOver=\"window.status=\'" +
                db[i].statusMsg + "\';return true;\">" + db[i].display +
                "</A><BR>"
        }
        prevIndentDisplayed = currIndent
        showMyDaughter = expanded
        if (db.length > 25) {
            document.write(newOutline)
            newOutline = ""
        }
    }
}
```

First, you call upon two previously defined functions to grab the widget image object and corresponding `onMouseOver` message for the statusbar. Two more variables contain the indent property for the item (that is, how many steps indented the item will appear in the outline structure) and the current expanded state, based on the cookie's entry for that item.

Not every entry in the outline database is displayed. For instance, a nested item whose mother is collapsed won't need to be displayed. To find out if an entry should be displayed, the script performs a number of tests on some of its values. An item can be displayed if any of the following conditions are met:

* The item is a topmost item, with an indentation factor of 0.

* The item is at the same or smaller indentation level as the previous item displayed.

* The previous item was tagged as being expanded, and the current item is indented from the previous item by one level.

Over the next few statements, the script pieces together the HTML for the outline entry, starting with the width necessary for the transparent filler image (based on the number of pixels specified for indentations near the top of the script). Next comes the link definition that wraps around the widget image. The following concepts apply to each link:

* The HREF attribute is the `javascript:` URL to invoke the `history.go()` method.

* The `onMouseOver` event handler is set to adjust the status message to the previously retrieved message (notice the `return true` statement to make the setting take effect).

* The `onClick` event handler is set to call the `toggle()` function, passing the number of the item within the outline database. An `onClick` event handler is carried out before the browser responds to the click of the link by navigating to the URL. Therefore, the `toggle()` function changes the setting of the cookie a fraction of a second before the browser refreshes the document (which relies on that new cookie setting). But click events on widgets that have no children do not need to hit the `toggle()` function. Therefore, the content of the `return` statement is influenced by whether or not the widget image is an endpoint image.

In the next statement, the `newOutline` string accumulation continues with the `<IMG>` tag specifications for the widget art. Specifying the `HEIGHT` and `WIDTH` attributes for the image is important, partly to help the browser lay out the page more quickly, partly to avoid pesky performance inconsistencies.

Next comes a decision about whether to display the item text as a link or as plain text. The script inspects the `db[i].URL` property to see if it is empty. If so, that means no URL is specified for a link, and the item should be built as plain text.

If a URL is specified for the item, the script instead constructs a link around the text. In this HTML assembly process, numerous calls to properties of the `db` array fetch properties of the entry for the URL, the statusbar message, and the text to display. Notice, too, that the link sets the target of the link to the frame name assigned to `displayTarget` near the top of the script.

As you near the end of the loop, two variable values, `prevIndentDisplayed` and `showMyDaughter`, are updated with settings from the current traversal through the loop. These values influence the display of nested items for the next entry's journey through the loop.

But before looping around again, the script inspects whether the outline is longer than 25 entries. If so, the script writes the outline entries that have accumulated so far, resetting the `newOutline` variable to empty for the next time through the loop. The reasoning behind this last routine is to help long outlines start to display their goods faster. I have seen Web site authors use this outline for literally hundreds of entries. At that quantity, the usually fast JavaScript begins to bog down a bit. By writing lines from a big outline to the page early, the user gets visual feedback that something is happening.

Once outside the loop, the script writes whatever last items may have accumulated in the `newOutline` variable. For outlines with less than 25 items, the whole outline is written in one push; for longer outlines, the value is empty at this point, because the intermediate writings have completed the job.

All that's left is to close up standard tags to finish the document definition:

```
document.write(newOutline)
```

```
// end -->
</SCRIPT>
</BODY>
</HTML>
```

Notice that the `document.write()` statement here is not followed by `document.close()`. Because this content is being written as the page loads, the output stream is closed at the end of the page's HTML.

Customization possibilities

Although this DHTML-free outliner is not the fanciest to be found on the Web, it is, nevertheless, quite popular probably due to its ease of customizability and backward compatibility to all but the earliest browsers (you can find the very original version at my Web site). Other page authors have pushed and pulled on this code to tailor it to a variety of special needs.

Alternative displays

At the root of almost all significant customization jobs lie modifications to the `dbRecord` object constructor near the beginning of the page and the HTML assembly portion in the Body. They work hand in hand. For example, one user wants different links in the outline to load pages into different targets. Most links are to load content into another frame of the same frameset, while others are to replace the frameset entirely. In the version provided previously, one target is assumed, and it is set as a global variable. But if you need to provide different targets for each item, you can add a new property (perhaps named `target`) to the `dbRecord` constructor, and assign the string name of the target (for example, "Frame2", "_top") to the property for each item. Then, in the HTML accumulation portion, assign the value of `db[i].target` to that `TARGET` attribute (watching out for the necessary pairings of quote symbols, as shown in other attribute assignments).

Another request asked that the text associated with the plus/minus images be clickable not to navigate to another page, but to expand and collapse the nested content. All the pieces for this variation are already in place. By performing minor reconstructive surgery on the HTML accumulator script, you can add a branch that looks for the `db[i].mother` property. If it’s `true`, then don’t write the closing `</A>` tag after the widget. Instead, branch to write the `db[i].display` text without its own URL link, and write the widget’s `</A>` tag after the text. Now the widget and text share the same link as the widget originally had.

Cookie-free zones

Not everyone likes to develop with cookies. That’s not a problem for this outliner, even though the previous example uses them liberally. The data that preserves the state of the outline is nothing but a string of `1`s and `0`s. If you are using a frameset, that string can be preserved as a global variable in the framesetting document.

To minimize the changes needed to the existing code, you can continue to use the same functions — `setCurrState()` and `getCurrState()` — as the interfaces to the reading and writing of the state. Begin by defining a global variable in the Head portion of the framesetting document, initializing it as an empty string:

```
<SCRIPT LANGUAGE="JavaScript">
outlineState = ""
</SCRIPT>
```

Now you can modify the two functions in the outliner page as follows:

```
// ** functions that get and set state data **
// set cookie data
var mycookie = document.cookie
function setCurrState(setting) {
        mycookie = parent.outlineState = setting
}
// retrieve cookie data
function getCurrState() {
        return parent.outlineState
}
```

Notice that there is no need for the label that has to be assigned to a cookie. The variable name keeps this data separate from the rest of the script space.

The only downside to not using a cookie is that the outline state is not preserved if the frameset goes away. If the user revisits the frameset in the same session, the outline state will be reinitialized at its beginning state.

Expanding/collapsing all at once

If you have an extensive outline, you may want to provide a shortcut to the user to expand everything at once or close up the entire outline. Because the string of `1`s and `0`s maintains the state of the outline, you can use the `db` array to help you create a new state string, and then apply it to the page. Here are two functions that do the job:

```
function expandAll() {
    expState = ""
    for (i = 1; i < db.length; i++) {
        expState += (db[i].mother) ? "1" : "0"
    }
    setCurrState(expState)
    history.go(0)
}

function collapseAll() {
    collState = ""
    for (i = 1; i < db.length; i++) {
        collState += "0"
    }
    setCurrState(collState)
    history.go(0)
}
```

All you need are a couple of buttons to invoke these functions, and you’re in business.

Reducing server access

Through the lifetime of this outliner application, it has seen wildly different behaviors of the various browsers with regard to how much the browser reaches out to the server for each redisplay of the outline. While the `history.go(0)` type of reloading is supposed to be the least onerous, some browsers seem to read the entire file from scratch. This approach is still faster than having a CGI script completely reconfigure a page, but for an extensive outline and a slow Internet connection, the results can be objectionable.

One possible solution is to avoid reloading the page at all. Instead, place all of the code for the outliner management and creation in the framesetting document. Code that currently writes the outline as the page loads can be encapsulated in a function that writes to the frame designated as the outline frame (don't forget the `document.close()` for this writing!). Function calls from the outliner (to `toggle()`, for instance) have to be modified so that the reference is to the function in the parent frame (`parent.toggle(`*`n`*`)`).

Distributing the code around frames may not be as convenient as keeping it all together, but user experience should weigh more heavily than programmer expedience. This practice also opens the possibility for putting all of the outliner code, except for the calls to the constructor functions, in an external `.js` library. You can then put multiple outline contents into multiple `.js` libraries, and load the pairs that you need into a frameset.

Using `document.write()` to another frame may still not avoid server access entirely. It is not uncommon for the application of any image file — including those that have been precached — to check the cached version against the modification date of the file on the server. This activity is much faster than downloading the image again, but if you see network activity even after shifting the outliner's scripts to the frameset, at least you understand what's happening. A version of the application directed from the parent window is contained on the CD-ROM.

Multiple outlines

The example in this chapter assumes that a site will be using only one outline-style table of contents. You can, of course, have multiple outlines for different sections of a Web site or application. But if the outlines all share the same cookie data, then the state of the most recent outline will be applied to the next one that loads. Items will be magically opened. And if the number of items between the two outlines is different, the cookie data can get a bit messy.

To solve this problem, assign a different cookie label for each outline. That prevents one outline's state from stepping on another.

Cascading Style Sheet Version

The advent of Cascading Style Sheets (CSS) brought a number of intriguing possibilities for an application, such as the outliner. Not only can style sheets be used to control the look of the items in the outline, but additional properties make it possible to hide and show elements, including inserting or removing elements from the rendered content. Alas, not all of these features work in NN4, so that the version under discussion in this section resorts to redrawing the outline for NN4. But for IE4+ and W3C DOMs, the response is very fast, and no page reloading is necessary. One of the goals, too, in this application was to reuse as much of the code from earlier versions as possible. Note that this version does not work (or work correctly) with browsers prior to NN4 or IE4.

CSS implementation plan

Many of the compromises in this version resulted from quirky behavior of NN4 with some types of elements and style sheets. I chose to render the outline content as a series of nested DIV elements. If this were being implemented strictly in more well behaved browsers, style sheet control over UL and LI elements would be even more convenient because those elements already have an indentation scheme built into them. With so much HTML code needed to generate the DIV elements and their contents, I decide to trade the cleverness of multidimensional array storage of outline content for the better performance of straight HTML. Each row of content in the outline is set in its own `<DIV>` block tag set. Any row that had children nested inside contains those items as a nested block.

Style sheets afforded the design a handy behavior. Hiding and showing blocks via the CSS-Positioning `visibility` property (see Chapter 30) is not an apt solution here, because hiding an item does not remove it from the page rendering. Therefore, unless the page included a ton of positioning code to overlap hidden items with visible items (which would have worked in NN4, but at the price of substantial increases in code and inflexibility), the outline would not cinch up if a branch is collapsed. To the rescue comes the `display` property of a style. One value of this property (`none`) not only hides the block, but it temporarily removes it from the rendering order of the page. Any items rendered below it that are visible (that is, whose `display` property is set to `block`) scoot up to render after the previous visible item.

Setting the `display` property has slightly different results in NN4 and more modern DOMs. In NN4, you can set the property after the block has been rendered on the page, but its appearance does not change; in both the IE4+ and W3C DOMs, the change is immediate, with the rest of the page reflowing to adjust to the change in the block’s visibility and presence. Therefore, for NN4, the page still needs to reload itself and remember the state of the outline between reloads (via the same cookie mechanism used for the earlier version) so that the page can set the property value as the page loads. And except for only a couple of places in the code, both the IE4+ and W3C DOMs share positioning code.

The CSS version uses the same cookie value (a sequence of `1` and `0` values) to represent the visible or hidden state of each item as in the old version. To convey the change of state, however, the function called by the click of an icon widget must pass the index values of the child items affected by the expansion or collapse of a node. This means that more of the HTML — in this case, the parameters of the functions — has to be hard-wired to the structure of the outline, as you see shortly. Less of this would be necessary if NN4's implementation of CSS offered the same level of scriptable introspection into HTML elements as IE4's implementation: We'd be able to employ the style property inheritance behavior to simplify the way blocks are shown and hidden. Because the two classes of browsers supported in this example are so different in this regard, the scripting reflects the lowest common denominator for controlling the toggle of expanded and collapsed states.

The CSS code

By putting so much of the content directly into HTML, the scripting component of the CSS outliner version is significantly smaller than the older version. Where possible, I stayed with the same function and variable naming schemes of the previous version.

At the top of the document, I define three styles for the amount of indentation required by the three indentation levels of my sample outline. If the outline were to go to more levels, I would add styles accordingly.

```
<HTML>
<HEAD>
<STYLE TYPE="text/css">
    DIV.indent0 {margin-left:0}
    DIV.indent1 {margin-left:10}
    DIV.indent2 {margin-left:20}
</STYLE>
```

Scripting begins by setting some global variables. Browser-specific branching comes into play later, but in an effort to stamp out explicit version detection, the code here relies on object detection to set the requisite flags. Only browsers capable of the CSS style scripting needed here have a `document.styleSheets` property, so flags are set for the two supported browser classes. These flags are set here primarily as a convenience for writing branching code later. Rather than constantly retesting for the presence of the property, the global flags are shorter and marginally faster. Two more variables hold their respective browser class state values, with the NN4 version maintaining a copy of the cookie as a variable for performance reasons.

```
<SCRIPT LANGUAGE="JavaScript">
// global variables
var isNN4, isCSS, CSScurrState, NN4Cookie = document.cookie
if (document.styleSheets) {
    isCSS = true
    isNN4 = false
} else {
    isCSS = false
    isNN4 = true
}
```

To each of the cookie storage functions from the original version, I add a branch to handle the storage and retrieval of state data for CSS browsers, simply setting and getting the global variable. This may seem to be more indirect than is necessary, but it is essential to allow the reuse of many functions in other parts of the code so that those areas don't have to worry about browser platform. Notice that the label for this outline’s cookie is slightly different from that of the earlier version. This difference allows you to open both outliners with NN4 in the same session and not worry about one cookie value overlapping with the other.

```
// ** functions that get and set persistent data **
// set persistent data
function setCurrState(setting) {
    if (isNN4) {
        NN4Cookie = document.cookie = "currState2=" + escape(setting)
    } else {
        // for CSS, data is saved as a global variable instead of cookie
        CSScurrState = setting
    }
}

// retrieve persistent data
function getCurrState() {
    if (isCSS) {
        // for CSS, data is in global var instead of cookie
        return CSScurrState
    }
    var label = "currState2="
    var labelLen = label.length
    var cLen = NN4Cookie.length
    var i = 0
    while (i < cLen) {
        var j = i + labelLen
        if (NN4Cookie.substring(i,j) == label) {
            var cEnd = NN4Cookie.indexOf(";",j)
            if (cEnd ==    -1) {
                cEnd = NN4Cookie.length
            }
            return unescape(NN4Cookie.substring(j,cEnd))
        }
        i++
    }
    return ""
}
```

The `toggle()` function is called by the `onClick` event handler of the links surrounding the widget icon art in the outline. A variable number of parameters are passed to this function, so that the parameters are extracted and analyzed via the `arguments` property of the function. Both browsers with only a few small browser-specific branches use a great deal of the code. Inside the large `for` loop, a CSS branch dynamically changes the setting of the `style.display` property. For NN4, the page is reloaded after all changes to the cookie version of the state are saved. After the NN4 version goes off to reload the page, the CSS version swaps the image of the toggled widget. As a final touch, the window is given focus so that IE/Windows browsers lose the dotted rectangle around the clicked image.

```
// **function that updates persistent storage of state**
// toggles an outline mother entry, storing new value
```

```
function toggle() {
    var newString = ""
    var expanded, n
    // get all <DIV> tag objects in IE4/W3C DOMs
    if (document.all) {
        var allDivs = document.all.tags("DIV")
    } else if (document.getElementsByTagName) {
        var allDivs = document.getElementsByTagName("DIV")
    }
    var currState = getCurrState() // of whole outline
    // assemble new state string based on passed parameters
    for (var i = 0; i < arguments.length; i++) {
        n = arguments[i]
        expanded = currState.charAt(n) // of clicked item
        newString += currState.substring(0,n)
        newString += expanded ^ 1 // Bitwise XOR clicked item
        newString += currState.substring(n+1,currState.length)
        currState = newString
        newString = ""
        if (isCSS) {
            // dynamically change display style without reloading
            if (expanded == "0") {
                allDivs[n].style.display = "block"
            } else {
                allDivs[n].style.display = "none"
            }
        }
    }
    setCurrState(currState) // write new state back to cookie
    if (isNN4) {
        location.reload()
    }
    // swap images in CSS versions
    var img = document.images["widget" + (arguments[0]-1)]
    img.src = (img.src.indexOf("plus.gif") != -1) ?
        "minus.gif" : "plus.gif"
    window.focus()
}
```

A prerequisite for loading the page to begin with is setting the initial value of the state. This is the only part of the script that must be hard-wired based on the structure of the outline — string assigned to `initState` will be different with each outline. The goal here is to set each block assigned to the `indent0` style class to `1` while all others are set to `0`. These settings allow the first display of the outline to show all the root nodes, with all other items collapsed.

```
// initialize 'current state' storage field
if (!getCurrState()) {
    // must be hard-wired to outline structure with "1" for
    // each indent0 class item, "0" for all others
    initState = "1000010000"
    setCurrState(initState)
}
```

With the initial outline state saved in the above code, the following statements execute at load time to write a `<STYLE>` tag set for NN4. This tag sets the `display` property of all collapsed blocks to `none`. As you see in the HTML coming up, blocks are assigned `ID` attributes with the letter "a" followed by a sequence number starting with zero.

```
    // for Navigator 4, set display style for flagged IDs to 'none'
    // each time the page (re)loads
    if (isNN4) {
        document.write("<STYLE TYPE='text/css'>")
        var visState = getCurrState()
        for (var i = 0; i < visState.length; i++) {
            if (visState.charAt(i) == "0") {
                document.write("#a" + i + " {display:none}\n")
            }
        }
        document.write("</STYLE>")
    }
```

Initial settings of the `display` property for IE4+ can be done programmatically only after the document loads (the tags must exist before their properties can be adjusted). The following `init()` function is called from the `onLoad` event handler. Each browser class has a different set of initialization tasks. Both branches rely on the current state setting, so that value is retrieved just once. In the CSS branch, the `style.display` properties for hidden blocks are set to `none`. For NN4, on the other hand, the `style.display` properties are set as the page reloads, but this loop swaps the widget image for expanded blocks to the `minus.gif` version.

```
// for CSS, initialize flagged tags to style display = "none"
// for NN4, set affected images to minus.gif
function init() {
    var visState = getCurrState()
    if (isCSS) {
        for (var i = 0; i < visState.length; i++) {
            if (visState.charAt(i) == "0") {
                // branch for browser object capability
                if (document.all) {
                    document.all("a" + i).style.display = "none"
               } else if (document.getElementsByTagName) {
                    document.getElementById("a" + i).style.display = "none"
                }
            }
        }
    } else if (isNN4) {
        for (i = 0; i < visState.length; i++) {
            if (visState.charAt(i) == "1") {
                if (i+1 < visState.length && visState.charAt(i+1) == "1") {
                    if (document.images["widget" + i]) {
                        document.images["widget" + i].src = "minus.gif"
                    }
                }
            }
        }
    }
}
</SCRIPT>
</HEAD>

<BODY onLoad="init()">
<CENTER><H3>Composition of Selected Foods</H3><HR></CENTER>
```

Now begins the HTML that defines the content of the outline. For readability, I have formatted the `<DIV>` tag sets to follow the indentation of the outline data (this listing looks much better if you open the file from the CD-ROM in your text editor with word wrap turned off). Each tag includes a `CLASS` attribute pointing to a class defined in the first `<STYLE>` tag of the page. Each tag also includes an `ID` attribute whose name begins with the letter "a" and a sequential serial number, starting with zero. Navigator uses the `ID` attributes to help it assign `display` property settings during each reload.

Like the older version of the outliner, each entry includes an image (surrounded by a clickable link) and a text entry (which may or may not be a link to a document). The link around the image includes a `javascript:` URL for the `HREF` attribute. When a link is for a widget that is a mother item, the parameters to the `toggle()` function are the serial numbers of the immediate children IDs whose display properties are to be adjusted in the `toggle()` function. These passed items only need to be in the immediate children, because any of their children inherit the `display` property of their parents. For example, the first widget toggles items 1 and 2 (ids `a1` and `a2`). Item 2 happens to be a parent to items 3 and 4. But when the `display` property of item 2 is set to `none`, then none of its children (items 3 and 4) are displayed, no matter how their display properties are set.

IMG elements associated with each toggled DIV are named along similar lines, with the name starting with “widget” and the same serial number as the containing DIV. If you look at the end of the `toggle()` function again, you’ll see that the name for the IMG element is derived from the first parameter received by the `toggle()` function. That first parameter will always be one number higher than the serial number for the widget image to swap. To help you visualize the numbering scheme used within the example, the numbered identifiers and methods that relay associated numbers are shown in boldface.

```
<DIV CLASS=indent0 ID="a0">
    <A HREF="javascript:toggle(1,2)" onMouseOver=
    "status='Click to expand/collapse nested items';return true"
    onMouseOut="status='';return true">
    <IMG NAME="widget0" SRC="plus.gif" HEIGHT=12 WIDTH=12
BORDER=0></A>&nbsp;Peas<BR>
    <DIV CLASS=indent1 ID="a1">
        <A HREF="javascript:void(0)" onMouseOver=
        "status='No further items';return true"
        onMouseOut="status='';return true">
        <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
BORDER=0></A>&nbsp;<A HREF="foods.htm#boiled"
        TARGET=Frame2>Boiled</A><BR>
    </DIV>
    <DIV CLASS=indent1 ID="a2">
        <A HREF="javascript:toggle(3,4)" onMouseOver=
        "status='Click to expand/collapse nested items';return true"
        onMouseOut="status='';return true">
        <IMG NAME="widget2" SRC="plus.gif" HEIGHT=12 WIDTH=12
        BORDER=0></A>&nbsp;<A HREF="foods.htm#canned"
        TARGET=Frame2>Canned</A><BR>
        <DIV CLASS=indent2 ID="a3">
            <A HREF="javascript:void(0)"
            onMouseOver="status='No further items';return true"
            onMouseOut="status='';return true">
            <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
            BORDER=0></A>&nbsp;<A HREF="foods.htm#alaska"
            TARGET=Frame2>Alaska</A><BR>
        </DIV>
        <DIV CLASS=indent2 ID="a4">
            <A HREF="javascript:void(0)"
            onMouseOver="status='No further items';return true"
            onMouseOut="status='';return true">
            <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
            BORDER=0></A>&nbsp;<A HREF="foods.htm#losodium"
            TARGET=Frame2>Low-Sodium</A><BR>
        </DIV>
    </DIV>
</DIV>

<DIV CLASS=indent0 ID="a5">
    <A HREF="javascript:toggle(6)" onMouseOver=
    "status='Click to expand/collapse nested items';return true"
    onMouseOut="status='';return true">
    <IMG NAME="widget5" SRC="plus.gif" HEIGHT=12 WIDTH=12
    BORDER=0></A>&nbsp;Pickles<BR>
    <DIV CLASS=indent1 ID="a6">
        <A HREF="javascript:toggle(7,8,9)" onMouseOver=
        "status='Click to expand/collapse nested items';return true"
        onMouseOut="status='';return true">
        <IMG NAME="widget6" SRC="plus.gif" HEIGHT=12 WIDTH=12
        BORDER=0></A>&nbsp;<A HREF="foods.htm#cucumber"
        TARGET=Frame2>Cucumber</A><BR>
        <DIV CLASS=indent2 ID="a7">
            <A HREF="javascript:void(0)" onMouseOver=
            "status='Click to expand nested items';return true"
            onMouseOut="status='';return true">
            <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
            BORDER=0></A>&nbsp;<A HREF="foods.htm#dill"
            TARGET=Frame2>Dill</A><BR>
        </DIV>
        <DIV CLASS=indent2 ID="a8">
            <A HREF="javascript:void(0)"
            onMouseOver="status='No further items';return true"
            onMouseOut="status='';return true">
            <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
            BORDER=0></A>&nbsp;<A HREF="foods.htm#fresh"
            TARGET=Frame2>Fresh</A><BR>
        </DIV>
        <DIV CLASS=indent2 ID="a9">
            <A HREF="javascript:void(0)"
            onMouseOver="status='No further items';return true"
            onMouseOut="status='';return true">
            <IMG SRC="end.gif" HEIGHT=12 WIDTH=12
            BORDER=0></A>&nbsp;<A HREF="foods.htm#sour"
            TARGET=Frame2>Sour</A><BR>
        </DIV>
    </DIV>
</DIV>
</BODY>
</HTML>
```

The CSS version (for the identical outline content) is a slightly smaller file size than the older, compatible one, but not so big a difference as to influence your choice. Browser compatibility should be your number one criterion. Ease of modification for changing content and improved user experience for browsers following the CSS branch are tied in second.

A Futuristic (XML) Outline

As XML and its associated technologies head toward a solid standardized footing, the latest browsers available as this edition is being written provide mixed support for some of the key features of an ideal environment. As those issues are sorting themselves out, getting to know portions of XML through the IE5+/Windows XML data island features is possible. While it's not normally okay to embed XML in an HTML document (that is, the two designations specify unique document types), IE5+/Windows provides an `<XML>` tag, in which you can insert XML tags. Scripts can access the elements inside the XML data island, referencing those elements as child nodes of the XML element. See Chapter 33 for the reference material on the IE XML element.

Birth of an XML specification

Collapsible outlines provide convenient ways to organize hierarchical information all around us. You'd be hard-pressed to find a more active proponent of the outline than Dave Winer, CEO of UserLand Software, Inc. (`http://www.userland.com`). Dave is a veteran software developer, as well as author and outspoken Web publisher. His `www.scripting.com` Web site is a popular destination if you want to find out the latest Internet and computing technology "buzz."

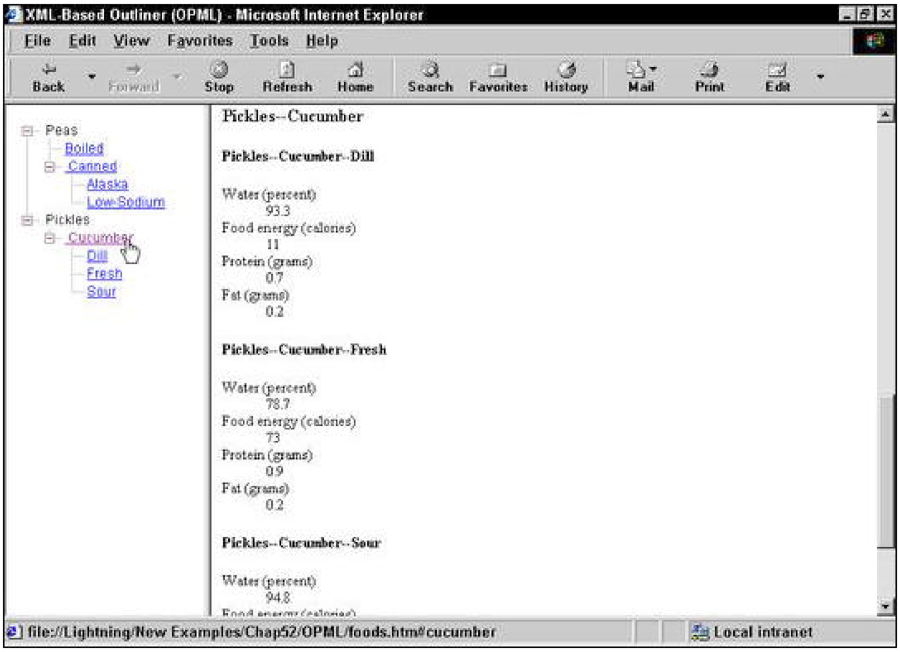
As an outgrowth of development for his company's Web tools, Dave looked to the XML structure to assist in representing outline content in a shareable, easily parseable format. The result is a specification called Outline Processor Markup Language, or OPML for short. You can read all about the formal specification at `http://www.opml.org/spec`. Like virtually all XML, OPML is intended to be written by software, not humans (although humans input the data via a user-friendly front-end provided by the software). Even so, the format of an OPML outline is extremely readable by humans, and, with little more trouble than writing basic HTML tags manually, you can represent an outline in this format yourself.

A plain OPML file, saved as an `.xml` file, can be viewed through the native XML parsers of IE5+ and NN6. These parsers automatically render XML tags in the same hierarchical fashion as OPML encourages outlines to be structured. But such rendering is under strict control of the browser, unless you also get involved with XML style sheets (the XSL and XSLT standards), at which point, browser implementation incompatibilities can make the going tough.

I liked the OPML data format when I first saw it, and I think it's a convenient way to convey an outline's data to the client, at which point JavaScript and the browser's DOM can take over to provide interesting visuals for the content and interaction with the content. Thus was born the last example of this chapter, in which the outliner's data is delivered not in the form of scripted arrays or hard-wired HTML DIV elements. Instead, the data arrives in its native XML (OPML) format inside an IE5+/Windows XML data island. Rendering of the native XML is suppressed, and scripts take over to do the rest.

OPML outliner prep

The appearance of widgets and text for the new outliner has changed to more closely emulate the kinds of outline presentations that you see in some Windows programs (see Figure 52-2). For demonstration purposes, the same frameset structure and outline content from earlier examples are used for the OPML version so that you can more easily see the differences in implementations and grasp new concepts presented here. For example, the comparison of how the outline data is delivered in the form of JavaScript objects (the first example) and OPML is enlightening.


XML-Based Outliner (OPML) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Peas
Boiled
Canned
Alaska
Low-Sodium
Pickles
Cucumber
Dill
Fresh
Sour
Local intranet

Pickles--Cucumber

Pickles--Cucumber--Dill

Water (percent)
93.3
Food energy (calories)
11
Protein (grams)
0.7
Fat (grams)
0.2

Pickles--Cucumber--Fresh

Water (percent)
78.7
Food energy (calories)
73
Protein (grams)
0.9
Fat (grams)
0.2

Pickles--Cucumber--Sour

Water (percent)
94.8
Food energy (calories)

file://Lightning/New Examples/Chap52/OPML/foods.htm#cucumber

Figure 52-2
OPML-based outliner style

As you recall, a custom object constructor function generated one JavaScript object for each outline entry. The properties of the object are completely under your control, so that you can add properties (such as the target of an entry's link), whose values influence the way the entry is rendered and the way it behaves. OPML has a similar extensibility feature. Each outline entry is nothing more than a tag. An entry that does not have any nested child nodes can use the XML shortcut of combining a start and end tag inside one set of angle brackets:

```
<tagName attribute="value" ... />
```

And any entry that has nested nodes contains the nested nodes between its start and end tags, as shown here with the actual tag names used in OPML (indentation is optional, but increases readability):

```
<outline text="text">
    <outline text="text"/>
    <outline text="text"/>
</outline
```

If you want to associate more information about an entry, simply add an attribute. For example, if an entry is to behave as a link, you can convey that information with an attribute whose name you determine. When it comes time for your scripts to render the content in HTML, the scripts access the attribute values and generate the associated HTML for the attributes (you see an example of this in the code).

The true beauty of the OPML structure (and XML in general) is that the parent–child relationships are automatically implied by the element containment. Unlike the JavaScript custom object in the first example, the author does not have to specify how many levels deep an entry is, or whether it has any child nodes: The XML containment hierarchy describes all of that information. Suddenly, all of the W3C DOM gobbledygook about nodes, child nodes, and attributes become your friend, as your scripts convert the element hierarchy into a renderable hierarchy of your design.

The XML and HTML code

Because our focus is so tight on the outliner content, you can start the exploration of the outliner code from the HTML BODY element downward, where the outline data is embedded in an IE5+/Windows XML element.

```
<BODY onLoad="init('outlineXML')">
<XML ID="outlineXML">
<opml version="1.0">
    <head>
        <title>A Modern Outline</title>
        <dateCreated>Thu, 16 Nov 2000 02:40:00 GMT</dateCreated>
        <dateModified>Fri, 22 Dec 2000 19:35:00 GMT</dateModified>
        <ownerName>Danny Goodman</ownerName>
        <ownerEmail>dannyg@dannyg.com</ownerEmail>
        <expansionState></expansionState>
        <vertScrollState>1</vertScrollState>
        <windowTop></windowTop>
        <windowLeft></windowLeft>
        <windowBottom></windowBottom>
        <windowRight></windowRight>
    </head>
    <body>
        <outline text="Peas">
            <outline text="Boiled" uri="foods.htm#boiled"/>
            <outline text="Canned" uri="foods.htm#canned">
                <outline text="Alaska" uri="foods.htm#alaska"/>
                <outline text="Low-Sodium" uri="foods.htm#losodium"/>
            </outline>
        </outline>
        <outline text="Pickles">
            <outline text="Cucumber" uri="foods.htm#cucumber">
                <outline text="Dill" uri="foods.htm#dill"/>
                <outline text="Fresh" uri="foods.htm#fresh"/>
                <outline text="Sour" uri="foods.htm#sour"/>
            </outline>
        </outline>
    </body>
</opml>
</XML>

<DIV ID="content"></DIV>
```

```
</BODY>
</HTML>
```

Everything inside the XML element is textbook OPML version 1.0 form. Notice that the OPML syntax re-uses element names that are found in all HTML files (for example, `head`, `title`, `body`). The XML island behavior isolates these tags from the browser's HTML rendering engine, so the browser won't confuse the two "documents." The only other HTML delivered in the document is an empty DIV element, which is used as the container for the outline HTML that the scripts generate as a result of the `onLoad` event handler's invocation of the `init()` function.

Also, go back to the top of the document to see the style sheets, which have an important place in delivering an XML island:

```
<HTML>
<HEAD>
<STYLE TYPE="text/css">
    XML {display:none}
    .row {vertical-align:middle; font-size:12px; font-family:Arial,sans-serif}
    .OLBlock {display:none}
    IMG {vertical-align:text-top}
</STYLE>
```

To prevent the XML block from rendering on the page, the `display` style property is set to `none` for the XML tag selector. This keeps the page clear for insertion of script-generated HTML. The other style sheet rules apply to content created by the scripts.

Setting the scripted stage

All scripts for this page are in the HEAD (although they could also be linked in from an external `.js` file). First on the docket is establishing several global variables that get used a lot within the rest of the code, and make it easy to customize important visible properties, especially widget art. Due to the art choices made for this version, there are separate versions for items that appear as first, middle, and end items for different nesting states.

```
<SCRIPT LANGUAGE="JavaScript">
// global variables
// art files and sizes for widget styles and spacers
// (all images must have same height/width)
var collapsedWidget = "oplus.gif"
var collapsedWidgetStart = "oplusStart.gif"
var collapsedWidgetEnd = "oplusEnd.gif"
var expandedWidget = "ominus.gif"
var expandedWidgetStart = "ominusStart.gif"
var expandedWidgetEnd = "ominusEnd.gif"
var nodeWidget = "onode.gif"
var nodeWidgetEnd = "onodeEnd.gif"
var emptySpace = "oempty.gif"
var chainSpace = "ochain.gif"
var widgetWidth = "20"
var widgetHeight = "16"
var currState = ""
var displayTarget = "Frame2"
```

The `init()` function, invoked by the `onLoad` event handler, starts the content creation in motion. The basic sequence is to first make sure that the browser is capable of recognizing an XML data island. If the validation is okay, then a reference to the BODY portion of the outline data is retrieved so that many other functions are able to dive into the outliner hierarchy. Notice that elements of the XML data island are disguised from view of the `document` object's normal scope. Access must be made by way of the XML object, which then exposes its elements. The reference to the OPML BODY element is passed to the `makeHTML()` function, which returns the entire outline HTML to be assigned to the `innerHTML` property of the empty DIV element delivered with the document.

```
// initialize first time
function init(outlineID) {
    if (supportVerified(outlineID)) {
        // demo how to get outline head elements
        var hdr =
document.getElementById(outlineID).getElementsByTagName("head")[0]
        // get outline body elements for iterative conversion to HTML
        var ol =
document.getElementById(outlineID).getElementsByTagName("body")[0]
        // wrap whole outline HTML in a span
        var olHTML = "<SPAN ID='renderedOL'>" +
            makeHTML(outlineID, ol) + "</SPAN>"
        // throw HTML into 'content' DIV for display
        document.getElementById("content").innerHTML = olHTML
        initExpand(outlineID)
    }
}
```

Validation of browser support is handled by the `supportVerified()` function. This function is in search of the `XMLDocument` property of the XML element object. The property's presence indicates that the browser has what it takes to treat embedded XML as a data island. Incremental tests are needed so that earlier browsers don't choke on the reference to the property.

```
// verify that browser supports XML islands
function supportVerified(testID) {
    if (document.getElementById &&
    document.getElementById(testID) &&
    document.getElementById(testID).XMLDocument) {
        return true
    } else {
        var reply = confirm("This example requires a browser with XML data island
support, such as IE5+/Windows. Go back to previous page?")
        if (reply) {
            history.back()
        } else {
            return false
        }
    }
    return false
}
```

Accumulating the HTML

From the `init()` function, a call to the `makeHTML()` function starts the most complex actions of the scripts on this page. This function walks the node hierarchy of the outline's BODY elements, deciphering which ones are containers and which ones are end points.

Two global variables are used to keep track of how far the node walk progresses because this function calls itself from time to time to handle nested branches of the node tree. Because a reflexive call to a function starts out with new values for local variables, the globals operate as pointers to let statements in the function know which node is being accessed. The numbers get applied to an `ID` attribute assigned to the DIV elements holding the content.

One of the fine points of the design of this outline is the way space to the left of each entry is assembled. Unlike the earlier outlines in this chapter, this one displays vertical dotted lines connecting nodes at the same level. There isn't a vertical line for every clickable node appearing above the item, because a clickable node may have no additional siblings, meaning that the space is blank. To see what I mean, open the OPML example, and expand the Peas and Canned nodes (or see Figure 52-2). The Canned node is the end of the second "column," so the space beneath the icon is blank. That's what some of the code in `makeHTML()` named "prefix" is dealing with: Accumulating just the right combination of dotted line (`chain.gif`) and blank (`empty.gif`) images in sequence before the outline entry.

Another frequent construction throughout this function is a three-level conditional expression. This construction is used to determine whether the image just to the left of the item's text should be a start, middle, or end version of the image. The differences among them are subtle (having to do with how the vertical dotted line extends above or below the widgets). All of these decisions are made from information revealed by the inherent structure of the OPML element nesting. The listing in the book looks longer than it truly is because so many long or deeply nested lines must be wrapped to the next line. Viewing the actual file in your text editor should calm your fears a bit.

```
// counters for reflexive calls to makeHTML()
var currID = 0
var blockID = 0
// generate HTML for outline
function makeHTML(outlineID, ol, prefix) {
    var output = ""
    var nestCount, link, nestPrefix
    prefix = (prefix) ? prefix : ""
    for (var i = 0; i < ol.childNodes.length ; i++) {
        nestCount = ol.childNodes[i].childNodes.length
        output += "<DIV CLASS='row' ID='line" + currID++ + "'>\n"
        if (nestCount > 0) {
            // for entries that are also parents
            output += prefix
            output += "<IMG ID='widget" + (currID-1) +
                "' SRC='" + ((i== ol.childNodes.length-1) ?
                collapsedWidgetEnd : (blockID==0) ?
                collapsedWidgetStart : collapsedWidget)
            output += "' HEIGHT=" + widgetHeight + " WIDTH=" +
                widgetWidth
            output += " TITLE='Click to expand/collapse nested items.'
                onClick='toggle(this," + blockID + ")'>"
            // if a uri is specified, wrap the text inside a link
            link =  (ol.childNodes[i].getAttribute("uri")) ?
                ol.childNodes[i].getAttribute("uri") : ""
            if (link) {
                output += "&nbsp;<A HREF='" + link +
                    "' CLASS='itemTitle' TITLE='" + link +
                    "' TARGET='" + displayTarget + "'>"
            } else {
                output += "&nbsp;<A CLASS='itemTitle' TITLE='" +
                    link + "'>"
            }
            // finally! the actual text of the entry
            output += "&nbsp;" + ol.childNodes[i].getAttribute("text") +
                "</A>"
            currState += calcBlockState(outlineID, currID-1)
            output += "<SPAN CLASS='OLBlock' BLOCKNUM='" + blockID +
                "' ID='OLBlock" + blockID++ + "'>"
            // accumulate prefix art for next indented level
            nestPrefix = prefix
            nestPrefix += (i == ol.childNodes.length - 1) ?
                "<IMG SRC='" + emptySpace + "' HEIGHT=16 WIDTH=20>" :
                "<IMG SRC='" + chainSpace + "' HEIGHT=16 WIDTH=20>"
            // reflexive call to makeHTML() for nested elements
            output += makeHTML(outlineID, ol.childNodes[i], nestPrefix)
            output += "</SPAN></DIV>\n"
        } else {
            // for endpoint nodes
            output += prefix
            output += "<IMG ID='widget" + (currID-1) + "' SRC='" +
                ((i == ol.childNodes.length - 1) ?
                nodeWidgetEnd : nodeWidget)
            output += "' HEIGHT=" + widgetHeight + " WIDTH=" +
                widgetWidth + ">"
            // check for links for these entries
            link =  (ol.childNodes[i].getAttribute("uri")) ?
                ol.childNodes[i].getAttribute("uri") : ""
            if (link) {
                output += "&nbsp;<A HREF='" + link +
                    "' CLASS='itemTitle' TITLE='" +
                link + "' TARGET='" + displayTarget + "'>"
            } else {
                output += "&nbsp;<A CLASS='itemTitle' TITLE='" +
                    link + "'>"
            }
            // grab the text for these entries
            output += ol.childNodes[i].getAttribute("text") + "</A>"
            output += "</DIV>\n"
        }
    }
   return output
}
```

As with the HTML assembly code of the first outliner, if you were to add attributes to OUTLINE elements in an OPML outline (for example, a URL for an icon to display in front of the text), it is in `makeHTML()` that the values would be read and applied to the HTML being created.

The only other function invoked by the `makeHTML()` function is `calcBlockState()`. This function looks into one of the OPML outline's HEAD elements, called EXPANSIONSTATE. This element's values can be set to a comma-delimited list of numbers corresponding to nodes that are to be shown expanded when the outline is first displayed. The `calcBlockState()` function is invoked for each parent element. The element's location is compared against values in the EXPANSIONSTATE element, if there are any, and returns the appropriate `1` or `0` value for the state string being assembled for the rendered outline.

```
// apply default expansion state from outline's header
// info to the expanded state for one element to help
// initialize currState variable
function calcBlockState(outlineID, n) {
    var ol = document.getElementById(outlineID).getElementsByTagName("body")[0]
    var outlineLen = ol.getElementsByTagName("outline").length
    // get OPML expansionState data
    var expandElem =
document.getElementById(outlineID).getElementsByTagName("expansionState")[0]
    var expandedData = (expandElem.childNodes.length) ?
        expandElem.firstChild.nodeValue.split(",") : null
    if (expandedData) {
       for (var j = 0; j < expandedData.length; j++) {
            if (n == expandedData[j] - 1) {
                return "1"
            }
        }
    }
    return "0"
}
```

The final act of the initialization process is a call to the `initExpand()` function. This function loops through the `currState` global variable (whose value was written in `makeHTML()` with the help of `calcBlockState()`) and sets the `display` property to `block` for any element designed to be expanded at the outset. HTML element construction in `makeHTML()` is performed in such a way that each parent DIV has a SPAN nested directly inside of it; and inside that SPAN are all the child nodes. The `display` property of the SPAN determines whether all of those children are seen or not.

```
// expand items set in expansionState XML tag, if any
function initExpand(outlineID) {
    for (var i = 0; i < currState.length; i++) {
        if (currState.charAt(i) == 1) {
            document.getElementById("OLBlock" + i).style.display = "block"
        }
    }
}
```

By the time the `initExpand()` function has run — a lot of setup code that executes pretty quickly — the rendered outline is in a steady state. Users can now expand or collapse portions by clicking the widget icons.

Toggling node expansion

All of the widget images in the outline have `onClick` event handlers assigned to them. The handlers invoke the `toggle()` function, passing parameters consisting of a reference to the IMG element object receiving the event and the serial number of the SPAN block nested just inside the DIV that holds the image. With these two pieces of information, the `toggle()` function sets in motion the act of inverting the expanded/collapsed state of the element and the plus or minus version of the icon image. The `blockNum` parameter corresponds to the position within the `currState` string of `1`s and `0`s holding the flag for the expanded state of the block. With the current value retrieved from `currState`, the value is inverted through `swapState()`. Then, based on the new setting, the `display` property of the block is set accordingly, and widget art is changed through two special-purpose functions.

```
// toggle an outline mother entry, storing new state value;
// invoked by onClick event handlers of widget image elements
function toggle(img, blockNum) {
    var newString = ""
    var expanded, n
    // modify state string based on parameters passed IMG
    expanded = currState.charAt(blockNum)
    currState = swapState(currState, expanded, blockNum)
    // dynamically change display style
    if (expanded == "0") {
        document.getElementById("OLBlock" + blockNum).style.display =
            "block"
        img.src = getExpandedWidgetState(img.src)
    } else {
        document.getElementById("OLBlock" + blockNum).style.display =
            "none"
        img.src = getCollapsedWidgetState(img.src)
    }
}
```

Swapping the state of the `currState` variable utilizes the same XOR operator employed by the first outliner in this chapter. The entire `currState` string is passed as a parameter. The indicated digit is segregated and inverted, and the string is reassembled before being returned to the calling statement in `toggle()`.

```
// invert state
function swapState(currState, currVal, n) {
    var newState = currState.substring(0,n)
    newState += currVal ^ 1 // Bitwise XOR item n
    newState += currState.substring(n+1,currState.length)
    return newState
}
```

As mentioned earlier, each of the clickable widget icons (plus and minus) can be one of three states, depending on whether the widget is at the start, middle, or end of a vertical-dotted chain. The two image swapping functions find out (by inspecting the URLs of the images currently occupying the IMG element) which version is currently in place so that, for instance, a starting plus (collapsed) widget is replaced with a starting minus (expanded) widget. This is a case of going the extra mile for the sake of an improved user interface.

```
// retrieve matching version of 'minus' images
function getExpandedWidgetState(imgURL) {
    if (imgURL.indexOf("Start") != -1) {
        return expandedWidgetStart
    }
    if (imgURL.indexOf("End") != -1) {
        return expandedWidgetEnd
    }
    return expandedWidget
}

// retrieve matching version of 'plus' images
function getCollapsedWidgetState(imgURL) {
    if (imgURL.indexOf("Start") != -1) {
        return collapsedWidgetStart
    }
    if (imgURL.indexOf("End") != -1) {
        return collapsedWidgetEnd
    }
    return collapsedWidget
}
```

Wrap up

There’s no question that the amount and complexity of the code involved for the OPML version of the outliner are significant. The “pain” associated with developing an application such as this is all up front. After that, the outline content is easily modifiable in the OPML format (or perhaps by some future editor that produces OPML output).

Even if you don’t plan to implement an OPML outline, the explanation of how this example works should drive home the importance of designing data structures that assist not only the visual design, but also the scripting that you use to manipulate the visual design.

Further Thoughts

The advent of CSS and element positioning has prompted numerous JavaScripters to develop another kind of hierarchical system of pop-up or drop-down menus. You can find examples of this interface at many of the JavaScript source Web sites listed in Appendix D. Making these kinds of menus work well in NN4, IE4+, and W3C DOMs is a lot of hard work, and if you can adopt code already ironed out by others, then all the better.

Most of the code you find, however, will require a fair amount of tweaking to blend the functionality into the visual design that you have or are planning for your Web site. Finding two implementations on the Web that look or behave the same way is rare. As long as you’re aware of what you’ll be getting yourself into, you are encouraged to check out these interface elements. By hiding menu choices except when needed, valuable screen real estate is preserved for more important, static content.