Chapter 55: Application: Decision Helper

In This Chapter

Multiple frames

Multiple-document applications

Multiple windows

Persistent storage (cookies)

Scripted image maps

Scripted charts

The list of key concepts for this chapter's application looks like the grand finale to a fireworks show. As JavaScript implementations go, the application is, in some respects, over the top, yet not out of the question for presenting a practical interactive application on a Web site without any server programming.

The Application

I wanted to implement a classic application often called a *decision support system*. My experience with the math involved here goes back to the first days of Microsoft Excel. Rather than design a program that had limited appeal (covering only one possible decision tree), I set out to make a completely user-customizable decision helper. All the user has to do is enter values into fields on a series of screens; the program performs the calculations to let the user know how the various choices rank against each other.

Although I won't be delving too deeply into the math inside this application, you will find it helpful to understand how a user approaches this program and what the results look like. The basic scenario is a user who is trying to evaluate how well a selection of choices measure up to his or her expectations of performance. User input includes:

* The name of the decision

* The names of up to five alternatives (people, products, ideas, and so on)

* The factors or features of concern to the user

* The importance of each of the factors to the user

* A user ranking of the performance of every alternative in each factor

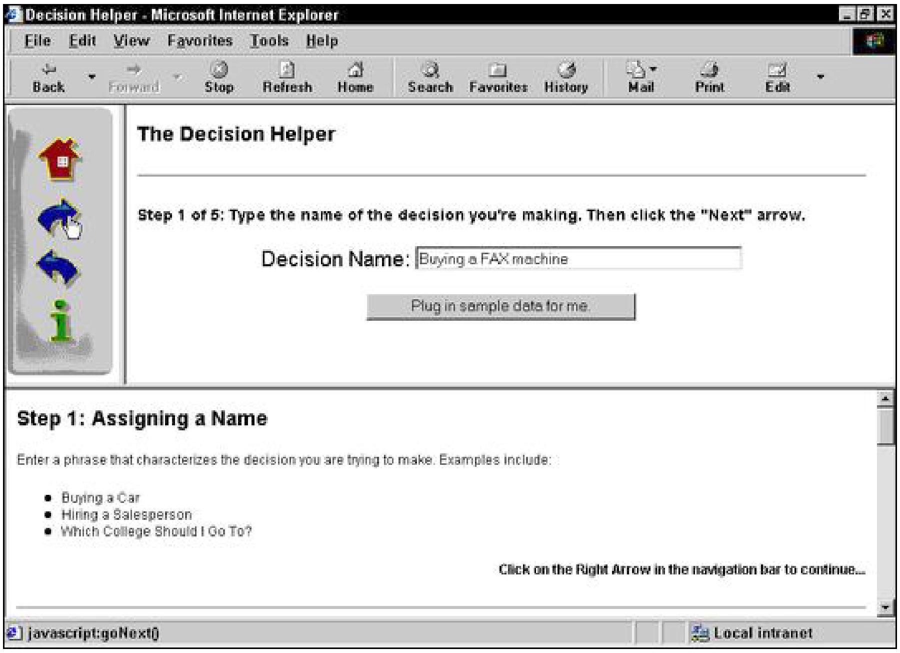
What makes this kind of application useful is that it forces the user to rate and weigh a number of often-conflicting factors. By assigning hard numbers to these elements, the user leaves the difficult process of figuring out the weights of various factors to the computer.

Results come in the form of floating-point numbers between 0 and 100. As an extra touch, I've added a graphical charting component to the results display.

The Design

With so much user input necessary for this application, conveying the illusion of simplicity was important. Rather than lump all text objects on a single scrolling page, I decided to break them into five pages, each consisting of its own HTML document. As an added benefit, I could embed information from early screens into the HTML of later screens, rather than having to create all changeable items out of text objects so that the application would work with older browsers. This "good idea" presented one opportunity and one rather large challenge.

The opportunity was to turn the interface for this application into something resembling a multimedia application using multiple frames. The largest frame would contain the forms the user fills out as well as the results page. Another frame would contain a navigation panel with arrows for moving forward and backward through the sequence of screens, plus buttons for going back to a home page and getting information about the program. I also thought a good idea would be to add a frame that provides instructions or suggestions for the users at each step. And so, the three-frame window was born, as shown in the first entry screen in Figure 55-1.


Decision Helper - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit

The Decision Helper

Step 1 of 5: Type the name of the decision you're making. Then click the "Next" arrow.

Decision Name: Buying a FAX machine

Plug in sample data for me.

Step 1: Assigning a Name

Enter a phrase that characterizes the decision you are trying to make. Examples include:

- Buying a Car
- Hiring a Salesperson
- Which College Should I Go To?

Click on the Right Arrow in the navigation bar to continue...

javascript:goNext()

Local intranet

Figure 55-1
The Decision Helper window consists of three frames.

Using a navigation bar also enables me to demonstrate how to script a client-side image map — not an obvious task with JavaScript.

On the challenge side of this design, finding a way to maintain data globally as the user navigates from screen to screen was necessary. Every time one of the entry pages unloads, none of its text fields is available to a script. My first attack at this problem was to store the data as global variable data (mostly arrays) in the parent document that creates the frames. Because JavaScript enables you to reference any parent document's object, function, or variable (by preceding the reference with `parent`), I thought this task would be a snap. A nasty bug in Navigator 2 (the prominent browser when this application was first developed) got in the way at the time: If a document in any child window unloaded, the variables in the parent window got jumbled. The other hazard here is that a reload of the frameset could erase the current state of those variables.

My next hope was to use the `document.cookie` as the storage bin for the data. A major problem I faced was that this program needs to store a total of 41 individual data points, yet no more than 20 uniquely named cookies can be allotted to a given domain. But the cookie proved to be the primary solution for this application (although see the "Further Thoughts" section at the end of the chapter about a non-cookie version on your CD-ROM). For some of the data points (which are related in an array-like manner), I fashioned my own data structures so that one named cookie could contain up to five related data points. That reduced my cookie demands to 17.

The Files

Before I get into the code, let me explain the file structure of this application. Table 55-1 gives a rundown of the files used in the Decision Helper.

Table 55-1
Files Comprising the Decision Helper Application

File	Description
`index.htm`	Framesetting parent document
`dhNav.htm`	Navigation bar document which contains some scripting
`dhNav.gif`	Image displayed in `dhNav.htm`
`dh1.htm`	First Decision Helper entry page
`dh2.htm`	Second Decision Helper entry page
`dh3.htm`	Third Decision Helper entry page

`dh4.htm`	Fourth Decision Helper entry page
`dh5.htm`	Results page
`chart.gif`	Tiny image file used to create bar charts in `dh5.htm`
`dhHelp.htm`	Sample data and instructions document for lower-right frame
dhAbout.htm	Document that loads into a second window

A great deal of interdependence exists among these files. As you see later, assigning the names to some of these files was strategic for the implementation of the image map.

The Code

With so many JavaScript-enhanced HTML documents in this application, you can expect a great deal of code. To best grasp what's going on here, first try to understand the structure and interplay of the documents, especially the way the entry pages rely on functions defined in the parent document. My goal in describing this structure is not to teach you how to implement this application, but rather how to apply the lessons I learned while building this application to the more complex ideas that may be aching to get out of your head and into JavaScript.

index.htm

Taking a top-down journey through the JavaScript and HTML of the Decision Helper, start at the document that loads the frames. Unlike a typical framesetting document, however, this one contains JavaScript code in its Head section — code that many other documents rely on.

```
<HTML>
<HEAD>
<TITLE>Decision Helper</TITLE>
```

An important consideration to remember is that in a multiple-frame environment, the title of the parent window's document is the name that appears in the window's title bar, no matter how many other documents are open inside its subframes.

The first items of the script control a global variable, `currTitle`, which is set by a number of the subsidiary files as the user navigates through the application. This variable ultimately helps the navigation bar buttons do their jobs correctly. Because this application relies on the `document.cookie` so heavily, the cookie management functions (slightly modified versions of Bill Dortch's Cookie Functions — Chapter 18) are located in the parent document so they load only once. I simplified the cookie writing function because this application uses default settings for pathname and expiration. With no expiration date, the cookies don't survive the current browser session, which is perfect for this application.

```
<SCRIPT LANGUAGE="JavaScript">
<!-- start
// global variable settings of current dh document number
var currTitle = ""
function setTitleVar(titleVal) {
    currTitle = "" + titleVal
}
function getCookieVal (offset) {
    var endstr = mycookie.indexOf (";", offset)
    if (("" + endstr) == "" || endstr == -1)
        endstr = mycookie.length
    return unescape(mycookie.substring(offset, endstr))
}

function getCookie (name) {
    var arg = name + "=";
    var alen = arg.length;
    var clen = mycookie.length;
    var i = 0;
    while (i < clen) {
        var j = i + alen;
        if (mycookie.substring(i, j) == arg) {
            return getCookieVal (j);
        }
        i = mycookie.indexOf(" ", i) + 1;
        if (i == 0) break;
    }
    return null;
}

var mycookie = document.cookie
function setCookie (name, value) {
    mycookie = document.cookie = name + "=" + escape (value) + ";"
}
```

When this application loads (or a user elects to start a new decision), it’s important to grab the cookies you need and initialize them to basic values that the entry screens will use to fill entry fields when the user first visits them. All statements inside the `initializeCookies()` function call the `setCookie()` function, defined in the preceding listing. The parameters are the name of each cookie and the initial value — mostly empty strings. Before going on, study the cookie labeling structure carefully. I refer to it often in discussions of other documents in this application.

```
function initializeCookies() {
    setCookie("decName","")
    setCookie("alt0","")
    setCookie("alt1","")
    setCookie("alt2","")
    setCookie("alt3","")
    setCookie("alt4","")
    setCookie("factor0","")
    setCookie("factor1","")
    setCookie("factor2","")
    setCookie("factor3","")
    setCookie("factor4","")
    setCookie("import","0")
    setCookie("perf0","")
    setCookie("perf1","")
    setCookie("perf2","")
    setCookie("perf3","")
    setCookie("perf4","")
}
```

The following functions should look familiar to you. They were borrowed either wholesale or with minor modification from the data-entry validation section of the Social Security number database lookup in Chapter 50. I'm glad I wrote these as generic functions, making them easy to incorporate into this script. Because many of the entry fields on two screens must be integers between 1 and 100, I brought the data validation functions to the parent document rather than duplicating them in each of the subdocuments.

```
// JavaScript sees numbers with leading zeros as octal values, so
// strip zeros
function stripZeros(inputStr) {
    return (parseFloat(inputStr, 10)).toString()
}

// general purpose function to see if a suspected numeric input
// is a positive integer
function isNumber(inputStr) {
    for (var i = 0; i < inputStr.length; i++) {
        var oneChar = charAt(i)
        if (oneChar < "0" || oneChar > "9") {
            return false
        }
    }
    return true
}

// function to determine if value is in acceptable range for this
// application
function inRange(inputStr) {
    num = parseInt(inputStr)
    if (num < 1 || num > 100) {
        return false
    }
    return true
}
```

To control the individual data-entry validation functions in the master controller, I again was able to borrow heavily from the application in Chapter 50.

```
// Master value validator routine
function isValid(inputStr) {
    if (inputStr != "" ) {
        inputStr = stripZeros(inputStr)
        if (!isNumber(inputStr)) {
            alert("Please make sure entries are numbers only.")
            return false
        } else {
            if (!inRange(inputStr)) {
                alert("Entries must be numbers between 1 and 100.  Try another
value.")
                return false
            }
        }
    }
    return true
}
```

Each of the documents containing entry forms retrieves and stores information in the cookie. Because all cookie functions are located in the parent document, it simplifies coding in the subordinate documents to have functions in the parent document acting as interfaces to the primary cookie functions. For each category of data stored as cookies, the parent document has a pair of functions for getting and setting data. The calling statements pass only the data to be stored when saving information; the interface functions handle the rest, such as storing or retrieving the cookie with the correct name.

In the following pair of functions, the decision name (from the first entry document) is passed back and forth between the cookie and the calling statements. Not only must the script store the data, but if the user returns to that screen later for any reason, the entry field must retrieve the previously entered data.

```
function setDecisionName(str) {
    setCookie("decName",str)
}
function getDecisionName() {
    return getCookie("decName")
}
```

The balance of the storage and retrieval pairs does the same thing for their specific cookies. Some cookies are named according to index values (`factor1`, `factor2`, and so on), so their cookie-accessing functions require parameters for determining which of the cookies to access, based on the request from the calling statement. Many of the cookie retrieval functions are called to fill in data in tables of later screens during the user's trip down the decision path.

```
// values for alternatives
function setAlternative(i,str) {
    setCookie("alt" + i,str)
}
function getAlternative(i) {
    return getCookie("alt" + i)
}

// values for decision factors
function setFactor(i,str) {
    setCookie("factor" + i,str)
}
function getFactor(i) {
    return getCookie("factor" + i)
}

// values for importance (decision factor weights)
function setImportance(str) {
    setCookie("import",str)
}
function getImportance(i) {
    return getCookie("import")
}

// values for performance ratings
function setPerformance(i,str) {
    setCookie("perf" + i,str)
}
function getPerformance(i) {
    return getCookie("perf" + i)
}
```

One sequence of code that runs when the parent document loads is the one that looks to see if a cookie structure is set up. If no such structure is set up (the retrieval of a designated cookie returns a `null` value), the script initializes all cookies via the function described earlier.

```
if (getDecisionName() == null) {
    initializeCookies()
}
// end -->
</SCRIPT>
</HEAD>
```

The balance of the parent document source code defines the frameset for the browser window. It establishes some hard-wired pixel sizes for the navigation panel. This assures that the entire `.gif` file is visible whenever the frameset loads, without a ton of unwanted white space if the browser window is large.

```
<FRAMESET ROWS="250,*">
    <FRAMESET COLS="104,*">
        <FRAME NAME="navBar" SRC="dhNav.htm" SCROLLING=no
        MARGINHEIGHT=2 MARGINWIDTH=1>
        <FRAME NAME="entryForms" SRC="dh1.htm">
    </FRAMESET>
    <FRAMESET ROWS="100%">
        <FRAME NAME="instructions" SRC="dhHelp.htm">
    </FRAMESET>
</FRAMESET>
</HTML>
```

I learned an important lesson about scripting framesets along the way. Older browsers, especially NN through Version 4, do not respond to changes in framesetting size attributes through a simple reload of the page. I found it necessary to reopen the frameset file from time to time. I also found it necessary to sometimes quit early Navigators altogether and relaunch it to make some changes visible. Therefore, if you seem to be making changes, but reloading the frameset doesn’t make the changes appear, try reopening or — as a last resort — quitting the browser.

dhNav.htm

Because of its crucial role in controlling the activity around this program, look into the navigation bar’s document next. To accomplish the look and feel of a multimedia program, this document was designed as a client-side image map that has four regions scripted corresponding to the locations of the four buttons (see Figure 55-1). One function is connected to each button.

The first function is linked to the graphical Home button. For the listing here, I just present an alert dialog box replicating the action of navigating back to a real Web site’s home page.

```
<HTML>
<HEAD>
<TITLE>Navigation Bar</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
function goHome() {
    alert("Navigate back to home page on a real site.")
}
```

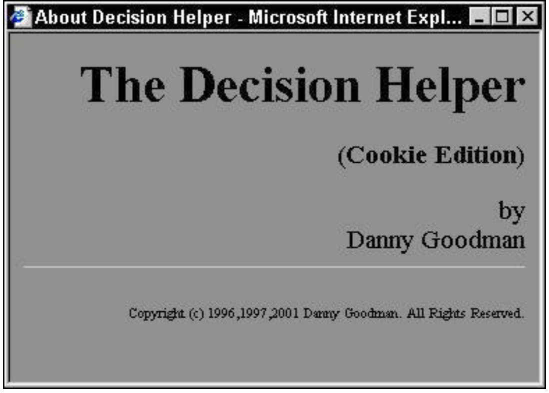
Each of the arrow navigation buttons brings the user to the next or previous entry screen in the sequence. To facilitate this without building tables of document titles and names, you call upon the `currTitle` global variable in the parent document. This value contains an integer in the range between 1 and 5, corresponding to the main content documents, `dh1.htm`, `dh2.htm`, and so on. As long as the offset number is no higher than the next-to-last document in the sequence, the `goNext()` function increments the index value by one and concatenates a new location for the frame. At the same time, the script advances the help document (in the bottom frame) to the anchor corresponding to the chosen entry screen by setting the `location.hash` property of that frame. Similar action navigates to the previous screen of the sequence through the `goPrev()` function. This time, the index value is decremented by one, and an alert warns the user if the current page is already the first in the sequence.

```
function goNext() {
    var currOffset = parseInt(parent.currTitle)
    if (currOffset <= 4) {
        ++currOffset
        parent.entryForms.location.href = "dh" + currOffset + ".htm"
        parent.instructions.location.hash = "help" + currOffset
    } else {
        alert("This is the last form.")
    }
}
function goPrev() {
    var currOffset = parseInt(parent.currTitle)
    if (currOffset > 1) {
        --currOffset
        parent.entryForms.location.href = "dh" + currOffset + ".htm"
        parent.instructions.location.hash = "help" + currOffset
    } else {
        alert("This is the first form.")
    }
}
```

Clicking the Info button displays a smaller window containing typical About-box data for the program (see Figure 55-2).

```
function goInfo() {
    var newWindow =
        window.open("dhAbout.htm","","HEIGHT=250,WIDTH=300")
}
// end -->
</SCRIPT>
</HEAD>
```

The Body of the navigation document contains the part that enables you to script a client-side image map. Mouse `click` events weren't available to AREA elements until Version 4 browsers, so to let these image maps work with older versions, mouse action is converted to script action by assigning a `javascript:` pseudo-URL to the `HREF` attribute for each AREA element. Instead of pointing to an entirely new URL (as AREA elements usually work), the attributes point to the JavaScript functions defined in the Head portion of this document. After a user clicks the rectangle specified by an `<AREA>` tag, the browser invokes the function instead.



The Decision Helper

(Cookie Edition)

by
Danny Goodman

Copyright (c) 1996,1997,2001 Danny Goodman. All Rights Reserved.

Figure 55-2
The About Decision Helper screen appears in a separate window.

```
<BODY>
<MAP NAME="navigation">
<AREA SHAPE="RECT" COORDS="23,22,70,67" HREF="javascript:goHome()">
<AREA SHAPE="RECT" COORDS="25,80,66,116" HREF="javascript:goNext()">
<AREA SHAPE="RECT" COORDS="24,125,67,161" HREF="javascript:goPrev()">
<AREA SHAPE="RECT" COORDS="35,171,61,211" HREF="javascript:goInfo()">
</MAP>
<IMG SRC="dhNav.gif" BORDER HEIGHT=240 WIDTH=96 ALIGN="left"
USEMAP="#navigation">
</BODY>
</HTML>
```

Although not shown here, you can assign `onMouseOver` event handlers to each AREA element for NN3+ and IE4+ to display a friendly message about the action of each button.

dh1.htm

Of the five documents that display in the main frame, `dh1.htm` is the simplest (refer to Figure 55-1). It contains a single entry field in which the user is invited to enter the name for the decision.

Only one function adorns the Head. This function summons one of the cookie interface functions in the parent window. A test is located here in case there is a problem with initializing the cookies. Rather than show `null` in the field, the conditional expression substitutes an empty string.

```
<HTML>
<HEAD>
<TITLE>DH1</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
function loadDecisionName() {
    var result = parent.getDecisionName()
    result = (result == null) ? "" : result
    document.forms[0].decName.value = result
}
// end -->
</SCRIPT>
</HEAD>
```

After the document loads, it performs three tasks (in the `onLoad` event handler). The first task is to set the global variable in the parent to let it know which number of the five main documents is currently loaded. Next, the script must fill the field with the decision name stored in the cookie. This task is important because users will want to come back to this screen to review what they entered previously. A third statement in the `onLoad` event handler sets the focus of the entire browser window to the one text object. This task is especially important in a multi-frame environment, such as this design. After a user clicks on the navigation panel, that frame has the focus. To begin typing into the field, the user has to tab (repeatedly) or click the text box to give the text box focus for typing. By setting the focus in the script when the document loads, you save the user time and aggravation.

```
<BODY onLoad="parent.setTitleVar(1);loadDecisionName();
document.forms[0].decName.focus()">
<H2>The Decision Helper</H2>
<HR>
<H4>Step 1 of 5: Type the name of the decision you're making. Then click the
"Next" arrow.</H4>
```

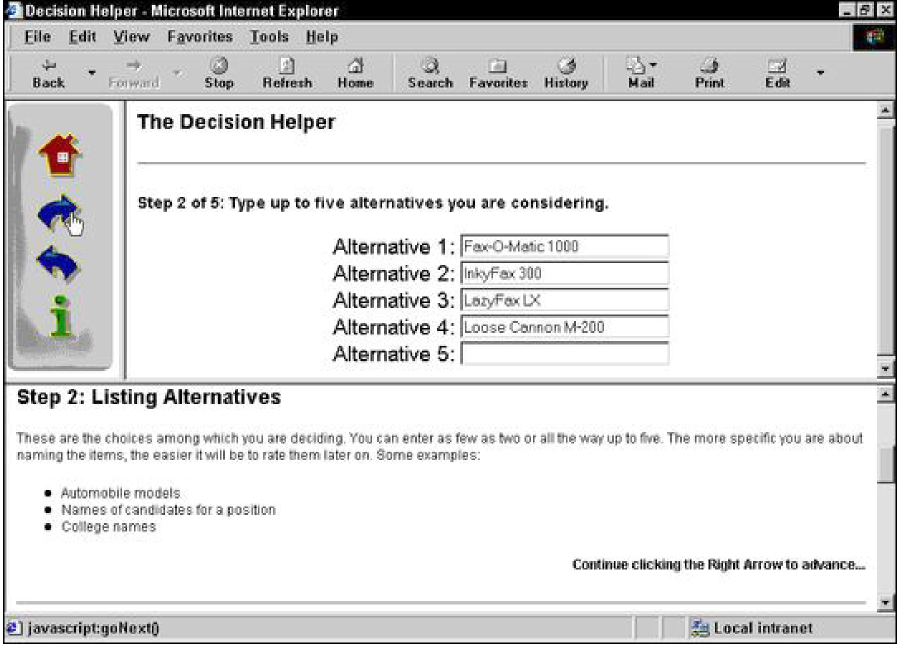
In the text field itself, an `onChange` event handler saves the value of the field in the parent's cookie for the decision name. No special Save button or other instruction is necessary here because any navigation that the user does via the navigation bar automatically causes the text field to lose focus and triggers the `onChange` event handler.

```
<CENTER>
<FORM>
Decision Name:
<INPUT TYPE="text" NAME="decName" SIZE="40"
onChange="parent.setDecisionName(this.value)">
</FORM>
</CENTER>
</BODY>
</HTML>
```

The copy of this file on the CD-ROM also has code that allows for plugging in sample data (as seen on my Web site) and a (commented out) textarea object that you can use for debugging cookie data.

dh2.htm

For the second data-entry screen (shown in Figure 55-3), five fields invite the user to enter descriptions of the alternatives under consideration. As with the decision name screen, the scripting for this page must both retrieve and save data displayed or entered in the fields.


Decision Helper - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit

The Decision Helper

Step 2 of 5: Type up to five alternatives you are considering.

Alternative 1: Fax-O-Matic 1000
Alternative 2: InkyFax 300
Alternative 3: LazyFax LX
Alternative 4: Loose Cannon M-200
Alternative 5:

Step 2: Listing Alternatives

These are the choices among which you are deciding. You can enter as few as two or all the way up to five. The more specific you are about naming the items, the easier it will be to rate them later on. Some examples:

- Automobile models
- Names of candidates for a position
- College names

Continue clicking the Right Arrow to advance...

javascript:goNext() Local intranet

Figure 55-3
The second data-entry screen

In one function, the script retrieves the alternative cookies (five total) and stuffs them into their respective text fields (as long as their values are not `null`). This function script uses a `for` loop to cycle through all five items — a common process throughout this application. Whenever a cookie is one of a set of five, the parent function has been written (in the following example) to store or extract a single cookie, based on the index value. Text objects holding like data (defined in the following listing) are all assigned the same name, so that JavaScript lets you treat them as array objects — greatly simplifying the placement of values into those fields inside a `for` loop.

```
<HTML>
<HEAD>
<TITLE>DH2</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
function loadAlternatives() {
    for (var i = 0; i < 5; i++) {
    var result = parent.getAlternative(i)
        result = (result == null) ? "" : result
        document.forms[0].alternative[i].value = result
    }
}
// end -->
</SCRIPT>
</HEAD>
```

After the document loads, the document number is sent to the parent's global variable, its fields are filled by the function defined in the Head, and the first field is handed the focus to assist the user in entering data the first time.

```
<BODY onLoad="parent.setTitleVar(2);loadAlternatives();
document.forms[0].alternative[0].focus()">
<H2>The Decision Helper</H2>
```

```
<HR>
<H4>Step 2 of 5: Type up to five alternatives you are considering.</H4>
```

Any change that a user makes to a field is stored in the corresponding cookie. Each `onChange` event handler passes its indexed value (relative to all like-named fields) plus the value entered by the user as parameters to the parent's cookie-saving function.

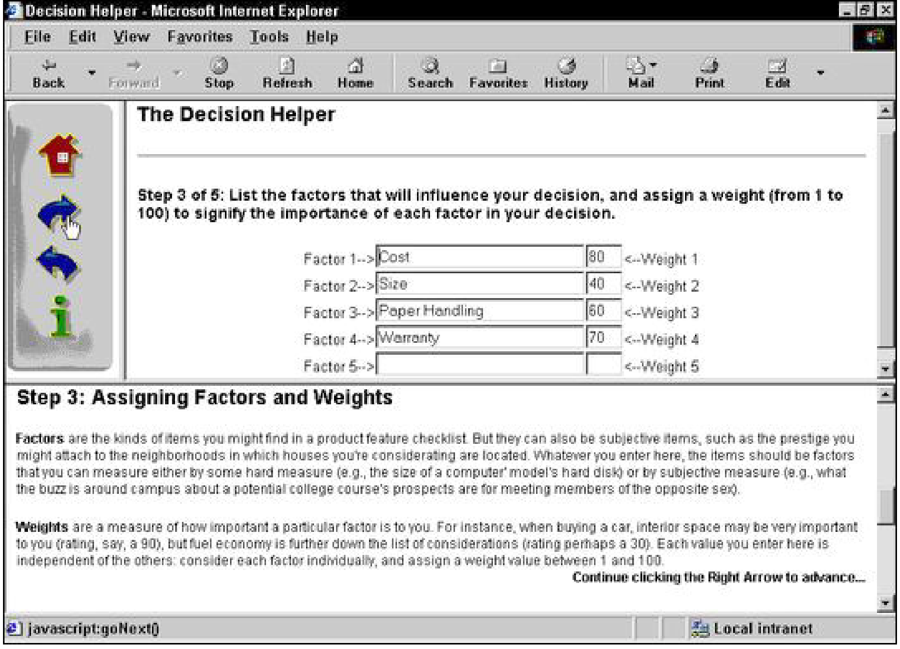
```
<CENTER>
<FORM>
Alternative 1:
<INPUT TYPE="text" NAME="alternative" SIZE="25"
onChange="parent.setAlternative(0,this.value)"><BR>
Alternative 2:
<INPUT TYPE="text" NAME="alternative" SIZE="25"
onChange="parent.setAlternative(1,this.value)"><BR>
Alternative 3:
<INPUT TYPE="text" NAME="alternative" SIZE="25"
onChange="parent.setAlternative(2,this.value)"><BR>
Alternative 4:
<INPUT TYPE="text" NAME="alternative" SIZE="25"
onChange="parent.setAlternative(3,this.value)"><BR>
Alternative 5:
<INPUT TYPE="text" NAME="alternative" SIZE="25"
onChange="parent.setAlternative(4,this.value)"><BR>
</BODY>
</HTML>
```

dh3.htm

With the third screen, the complexity increases a bit. Two factors contribute to this increase in difficulty. One is that the limitation on the number of cookies available for a single domain forces you to join into one cookie the data that might normally be distributed among five cookies. Second, with the number of text objects on the page (see Figure 55-4), it becomes more efficient (from the standpoint of tedious HTML writing) to let JavaScript deploy the fields. The fact that two sets of five related fields exist facilitates using `for` loops to lay out and populate them.

One initial function here is reminiscent of Head functions in previous entry screens. This function retrieves a single factor cookie from the set of five cookies.

```
<HTML>
<HEAD>
<TITLE>DH3</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
function getdh3Factor (i) {
    var result = parent.getFactor(i)
    if (result == null) {
        return ""
    }
    return result
}
```


Decision Helper - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit

The Decision Helper

Step 3 of 5: List the factors that will influence your decision, and assign a weight (from 1 to 100) to signify the importance of each factor in your decision.

Factor 1-->	Cost	80	<--Weight 1
Factor 2-->	Size	40	<--Weight 2
Factor 3-->	Paper Handling	60	<--Weight 3
Factor 4-->	Warranty	70	<--Weight 4
Factor 5-->			<--Weight 5

Step 3: Assigning Factors and Weights

Factors are the kinds of items you might find in a product feature checklist. But they can also be subjective items, such as the prestige you might attach to the neighborhoods in which houses you're considerating are located. Whatever you enter here, the items should be factors that you can measure either by some hard measure (e.g., the size of a computer' model's hard disk) or by subjective measure (e.g., what the buzz is around campus about a potential college course's prospects are for meeting members of the opposite sex).

Weights are a measure of how important a particular factor is to you. For instance, when buying a car, interior space may be very important to you (rating, say, a 90), but fuel economy is further down the list of considerations (rating perhaps a 30). Each value you enter here is independent of the others: consider each factor individually, and assign a weight value between 1 and 100.

Continue clicking the Right Arrow to advance...

javascript:goNext()   Local intranet

Figure 55-4
Screen for entering decision factors and their weights

Values for the five possible weight entries are stored together in a single cookie. To make this work, I had to determine a data structure for the five "fields" of a single cookie "record." Because all entries are integers, I can choose any nonnumeric character. I arbitrarily selected the period.

```
function setdh3Importance () {
    var oneRecord = ""
    for (var i = 0; i < 5; i++) {
        var dataPoint = document.forms[0].importance[i].value
        if (!parent.isValid(dataPoint)) {
            document.forms[0].importance[i].focus()
            document.forms[0].importance[i].select()
            return
        }
        oneRecord += dataPoint + "."
    }
    parent.setImportance(oneRecord)
    return
}
```

The purpose of the `setdh3Importance()` function is to assemble all five values from the five Weight entry fields (named "importance") into a period-delimited record that is ultimately sent to the cookie for safekeeping. Another of the many `for` loops in this application cycles through each of the fields, checking for validity and then appending the value with its trailing period to the variable (`oneRecord`) that holds the accumulated data. As soon as the loop finishes, the entire record is sent to the parent function for storage.

Although the function shows two `return` statements, the calling statement does not truly expect any values to be returned. Instead, I use the `return` statement inside the `for` loop as a way to break out of the `for` loop without any further execution whenever an invalid entry is found. Just prior to that, the script sets the focus and select to the field containing the invalid entry. JavaScript, however, is sensitive to the fact that a function with a `return` statement in one possible outcome doesn't have a `return` statement for other outcomes (an error message to this effect appears in some browsers if you try the function without balanced returns). By putting a `return` statement at the end of the function, all other possibilities are covered to the browser's satisfaction.

The inverse of storing the weight entries is retrieving them. Because the `parent.getImportance()` function returns the entire period-delimited record, this function must break apart the pieces and distribute them into their corresponding Weight fields. A combination of string methods determines the offset of the period and how far the data extraction should go into the complete record. Before the `for` loop repeats each time, it is shortened by one "field's" data. In other words, as the `for` loop executes, the copy of the cookie data returned to this function is pared down one entry at a time as each entry is stuffed into its text object for display.

```
function getdh3Importance () {
    var oneRecord = parent.getImportance()
    if (oneRecord != null) {
        for (var i = 0; i < 5; i++) {
            var recLen = oneRecord.length
            var offset = oneRecord.indexOf(".")
            var dataPoint = (offset >= 0 ) ?
                oneRecord.substring(0,offset) : ""
            document.forms[0].importance[i].value = dataPoint
            oneRecord = oneRecord.substring(offset+1,recLen)
        }
    }
}

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

Upon loading the document, the only tasks that the `onLoad` event handler need to do are to update the parent global variable about the document number and to set the focus to the first entry field of the form.

```
<BODY onLoad=" parent.setTitleVar(3);document.forms[0].factor[0].focus()">
<H2>The Decision Helper</H2>
<HR>
<H4>Step 3 of 5: List the factors that will influence your decision,
and assign a weight (from 1 to 100) to signify the importance of each factor in
your decision.</H4>
```

Assembling the HTML for the form and its ten data-entry fields needs only a few lines of JavaScript code. Performed inside a `for` loop, the script assembles each line of the form, which consists of a label for the Factor (and its number), the factor input field, the importance input field, and the label for the Weight (and its number). A `document.write()` method writes each line to the document.

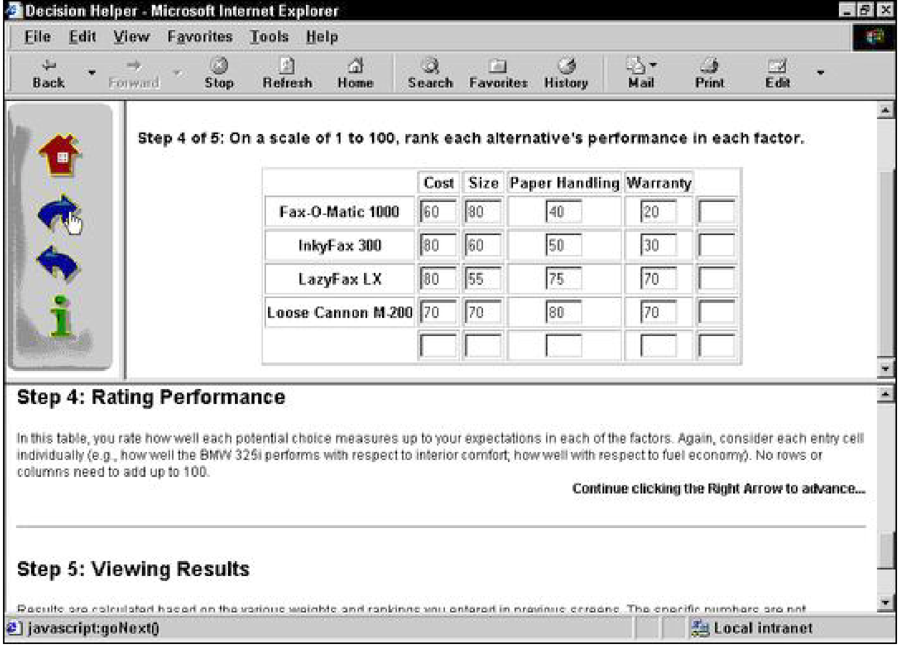
```
<SCRIPT LANGUAGE="JavaScript">
<!-- start
var output = "<CENTER><FORM>"
for (i = 0; i < 5; i++) {
    output += "Factor " + (i+1) +
        "--><INPUT TYPE='text' NAME='factor' SIZE='25' "
    var eHandler = " onChange=\'parent.setFactor(" + i + ",this.value)\'"
    output += eHandler + "VALUE=" + getdh3Factor (i) + ">"

    output += "<INPUT TYPE='text' NAME='importance' SIZE='3' "
    var eHandler = " onChange=\ setdh3Importance ()\'"
    output += eHandler + "VALUE=''>"
    output += "<--Weight " + (i+1) + "<BR>"
    document.write(output)
    output = ""
}
document.write("</FORM></CENTER>")
getdh3Importance ()
// end -->
</SCRIPT>
</BODY>
</HTML>
```

Each of the scripted text objects has an event handler. Notice that each event handler is first defined as a variable on a statement line just above its insertion into the string being assembled for the INPUT object definition. One reason for this fact is that the nested quote situation gets quite complex when you are doing these tasks all in one massive assignment statement. Rather than mess with matching several pairs of deeply nested quotes, I found it easier to break out one portion (the event handler definition) as a variable value and then insert that preformatted expression into the concatenated string for the INPUT definition.

Notice, too, how the different ways of storing the data in the cookies influence the ways the existing cookie data is filled into the fields as the page draws itself. For the factors, which have one cookie per factor, the `VALUE` attribute of the field is set with a specific indexed call to the parent factor cookie retriever, one at a time. But for the importance values, which are stored together in the period-delimited chunk, a separate function call (`getdh3Importance()`) executes after the fields are already drawn (with initial values of empty strings) and fills all the fields in a batch operation.

dh4.htm

Step 4 of the decision process (shown in Figure 55-5) is the most complex step because of the sheer number of entry fields: 25 in all. Notice that this screen retrieves data from two of the previous screens (or rather from the cookies preserving the entries) and embeds the values into the fixed parts of the table. All these tasks are possible when you create those tables with JavaScript.


Decision Helper - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit

Step 4 of 5: On a scale of 1 to 100, rank each alternative's performance in each factor.

	Cost	Size	Paper Handling	Warranty	
Fax-O-Matic 1000	60	80	40	20	
InkyFax 300	80	60	50	30	
LazyFax LX	80	55	75	70	
Loose Cannon M-200	70	70	80	70	

Step 4: Rating Performance

In this table, you rate how well each potential choice measures up to your expectations in each of the factors. Again, consider each entry cell individually (e.g., how well the BMW 325i performs with respect to interior comfort; how well with respect to fuel economy). No rows or columns need to add up to 100.

Continue clicking the Right Arrow to advance...

Step 5: Viewing Results

Results are calculated based on the various weights and rankings you entered in previous screens. The specific numbers are not

javascript:goNext()

Local intranet

Figure 55-5
A massive table includes label data from earlier screen entries.

Functions for getting and setting performance data are complex because of the way I was forced to combine data into five "field" records. In other words, one parent cookie exists for each row of data cells in the table. To extract cell data for storage in the cookie, I use nested `for` loop constructions. The outer loop counts the rows of the table, whereas the inner loop (with the `j` counter variable) works its way across the columns for each row.

Because all cells are named identically, they are indexed with values from 0 to 24. Calculating the row (`i * 5`) plus the column number establishes the cell index value. After you check for validity, each cell's value is added to the row's accumulated data. Each row is then saved to its corresponding cookie. As in the code for `dh3.htm`, the `return` statement is used as a way to break out of the function if an entry is deemed invalid.

Retrieving the data and populating the cells for the entire table requires an examination of each of the five performance cookies, and for each labeled cookie's data, a parsing for each period-delimited entry. After a given data point is in hand (one entry for a cell), it must go into the cell with the proper index.

```
<HTML>
<HEAD>
<TITLE>DH4</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
function getdh4Performance () {
    var oneRecord = ""
    for (var i = 0; i < 5; i++) {
        oneRecord = parent.getPerformance(i)
        if (oneRecord == null) {
            continue
        }
        for (var j = 0; j < 5; j++) {
            var recLen = oneRecord.length
            var offset = oneRecord.indexOf(".")
            var dataPoint = oneRecord.substring(0,offset)
            var cellNum = j + (i * 5)
            document.forms[0].ranking[cellNum].value = dataPoint
            oneRecord = oneRecord.substring(offset+1,recLen)
        }
    }
}
// end -->
</SCRIPT>
</HEAD>
function setdh4Performance () {
    for (var i = 0; i < 5; i++) {
        var oneRecord = ""
        for (var j = 0; j < 5; j++) {
            var cellNum = j + (i * 5)
            var dataPoint = document.forms[0].ranking[cellNum].value
            if (!parent.isValid(dataPoint)) {
                document.forms[0].ranking[cellNum].focus()
                document.forms[0].ranking[cellNum].select()
                return
            }
            oneRecord += dataPoint + "."
        }
        parent.setPerformance(i,oneRecord)
    }
    return
}
```

After the document is loaded, the `onLoad` event handler sends the document number to the parent global variable and brings focus to the first field of the table.

```
<BODY
onLoad=" parent.setTitleVar(4);document.forms[0].ranking[0].focus()">
<H2>The Decision Helper</H2>
<HR>
<H4>Step 4: On a scale of 1 to 100, rank each alternative's
performance in each factor.</H4>
<P><P>
```

To lessen the repetitive HTML for all tables, JavaScript again assembles and writes the HTML that defines the tables. In the first batch, the script uses yet another `for` loop to retrieve the factor entries from the parent cookie so that the words can be embedded into `<TH>` tags of the first row of the table. If every factor field is not filled in, the table cell is set to empty.

```
<SCRIPT LANGUAGE="JavaScript">
<!-- start
var output = "<CENTER><FORM NAME='perfRankings'><TABLE BORDER>"
output += "<TR><TD></TD><TD>"
for (var i = 0; i < 5; i++) {
    var oneFactor = parent.getFactor(i)
    oneFactor = (oneFactor == null) ? "" : oneFactor
    output += "<TH>" + oneFactor + "</TH>"
}
output += "</TD>"
```

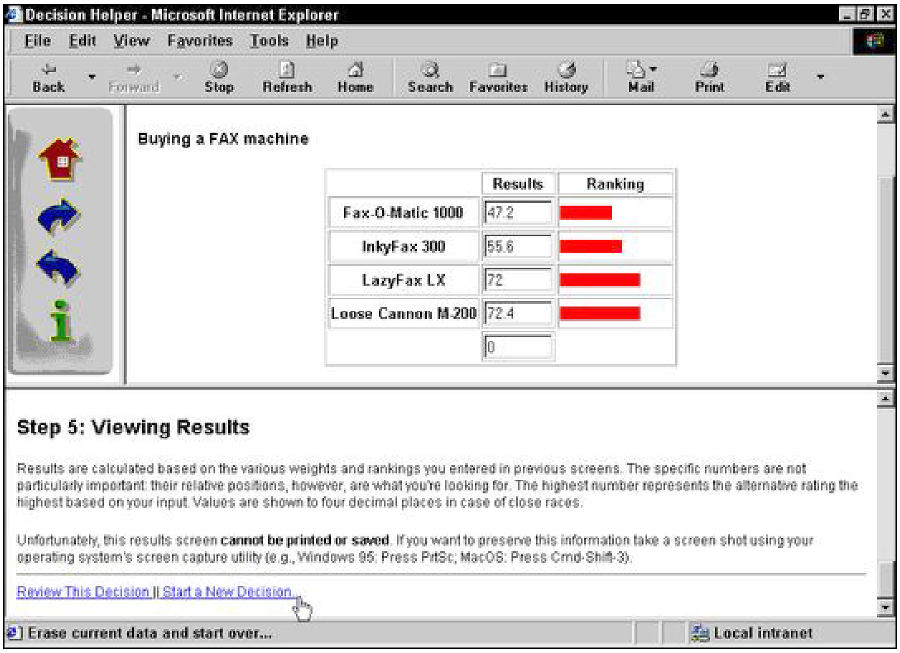
Next comes the assembly of subsequent rows of the table. The first column displays the name of each alternative (within `<TH>` tags). The remaining columns are text boxes, all with the same name and event handler. As each row of table definition is completed, it is written to the document. After the table and form closing tags are also written, the `getdh4Performance()` function retrieves all cookie data for the fields and distributes it accordingly.

```
for (var i = 0; i < 5; i++) {
    var oneAlt = parent.getAlternative(i)
    oneAlt = (oneAlt == null) ? "" : oneAlt
    output += "<TR><TD><TH>" + oneAlt + "</TH>"
    for (var j = 0; j < 5; j++) {
        output += "<TD ALIGN=CENTER><INPUT TYPE='text' SIZE=3 " +
        "NAME='ranking' VALUE='' onChange='setPerformance()'></TD>"
    }
    output += "</TR>"
    document.write(output)
    output = ""
}
document.write("</TABLE></FORM></CENTER>")
getdh4Performance ()
// end -->
</SCRIPT>
</BODY>
</HTML>
```

dh5.htm

From a math standpoint, dh5.htm's JavaScript gets pretty complicated. But because the complexity is attributed to the decision support calculations that turn the user's entries into results, I treat the calculation script shown here as a black box. You're free to examine the details, if you're so inclined.

Results appear in the form of a table (see Figure 55-6) with columns showing the numeric results and an optional graphical chart.



Decision Helper - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit

Buying a FAX machine

	Results	Ranking
Fax-O-Matic 1000	47.2	🟥
InkyFax 300	55.6	🟥
LazyFax LX	72	🟥
Loose Cannon M-200	72.4	🟥
	0	

Step 5: Viewing Results

Results are calculated based on the various weights and rankings you entered in previous screens. The specific numbers are not particularly important: their relative positions, however, are what you're looking for. The highest number represents the alternative rating the highest based on your input. Values are shown to four decimal places in case of close races.

Unfortunately, this results screen **cannot be printed or saved**. If you want to preserve this information take a screen shot using your operating system's screen capture utility (e.g., Windows 95: Press PrtSc; MacOS: Press Cmd-Shift-3).

Review This Decision || Start a New Decision...

Erase current data and start over...   Local intranet

Figure 55-6
The results screen for a decision

For the purposes of this example, you only need to know a couple of things about the `calculate()` function. First, this function calls all the numeric data stored in parent cookies to fulfill values in its formulas. Second, results are tabulated and placed into a five-entry indexed array called `itemTotal[i]`. This array is defined as a global variable, so that its contents are available to scripts coming up in the Body portion of the document.

```
<HTML>
<HEAD>
<TITLE>DH5</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
<!-- start
var itemTotal = new Array()
function calculate() {
    var scratchpad = ""
    var importanceSum = 0
    var oneRecord = parent.getImportance()
    var weight = new Array()
    for (var i = 0; i < 5; i++) {
        var recLen = oneRecord.length
        var offset = oneRecord.indexOf(".")
        scratchpad = oneRecord.substring(0,offset)
        importanceSum += (scratchpad == "" || scratchpad == "NaN") ?
            0 : parseInt(scratchpad)
        oneRecord = oneRecord.substring(offset+1,recLen)
    }
    oneRecord = parent.getImportance()
    for (var i = 0; i < 5; i++) {
        recLen = oneRecord.length
        offset = oneRecord.indexOf(".")
        scratchpad = oneRecord.substring(0,offset)
        weight[i] = (scratchpad == "" && scratchpad == "NaN") ?
            0 : parseInt(scratchpad)/importanceSum * 100
        oneRecord = oneRecord.substring(offset+1,recLen)
    }
    for (var i = 0; i < 5; i++) {
        oneRecord = parent.getPerformance(i)
        if (oneRecord == null) {
            continue
        }
        scratchpad = 0
        for (var j = 0; j < 5; j++) {
            var recLen = oneRecord.length
            var offset = oneRecord.indexOf(".")
            var dataPoint = oneRecord.substring(0,offset)
            scratchpad += (dataPoint != "" || dataPoint == "NaN") ?
                parseInt(dataPoint) * weight[j] / 100 : 0
            oneRecord = oneRecord.substring(offset+1,recLen)
        }
        itemTotal[i] = scratchpad
    }
}
calculate()
// end -->
</SCRIPT>
</HEAD>
```

Constructing this function served up many reminders about keeping data types straight. Because the data stored in cookies was in the form of strings, when it comes time to do some real math with those values, careful placement of the `parseInt()` function is essential for getting the math operators to work.

An `onLoad` event handler sends the document number to the global variable, as usual. The results display in this document relies heavily on stored and calculated values, so the table is constructed entirely out of JavaScript. That also means it can redisplay the decision name as part of the page.

```
<BODY onLoad="parent.setTitleVar(5)">
<H2>The Decision Helper</H2>
<HR>
<SCRIPT LANGUAGE="JavaScript">
<!-- start
document.write("<H4>" + parent.getDecisionName() + "</H4><P><P>")
var output = "<CENTER><FORM NAME='Results'><TABLE BORDER>"
output += "<TR><TD></TD><TD><TH>Results</TH><TH>Ranking</TH>"
output += "</TD>"
```

I need to break up the discussion of the `for` loop that produces the results because there are two distinct parts of this HTML assembly. The first, shown in the following script segment, assembles the first two cells of each row of the table. The first cell contains an embedded listing of the alternative name (in `<TH>` tags). To highlight the calculated values — and enable the `SIZE` attribute to do the artificial job of truncating the floating-point number — the results are shown in text boxes. For each row, the corresponding result in `itemTotal[i]` is inserted as the `VALUE` attribute of the text box. The `SIZE` attribute is set to 7, which allows the typical double-digit results, a decimal point, and four digits to the right of the decimal (an extra pixel shows on the Macintosh version, however).

```
for (var i = 0; i < 5; i++) {
    var oneAlt = parent.getAlternative(i)
    oneAlt = (oneAlt == null) ? "" : oneAlt
    itemTotal[i] = (oneAlt == "") ? 0 : itemTotal[i]
    output += "<TR><TD><TH>" + oneAlt + "</TH>"
    output += "<TD ALIGN=CENTER><INPUT TYPE='text' SIZE=7 " +
        "NAME='ranking' VALUE=" + itemTotal[i] + "></TD>"
```

For extra pizzazz, a third column “draws” a bar chart within a 100-pixel wide cell. The bars are actually scalings of a one-pixel-wide `.gif` file (an orange line, 12 pixels tall). A single-color `.gif` image scales to fill whatever width is assigned in the `WIDTH` attribute. This method is faster and far better than a more tedious method (tedious from the Web page author’s point of view) of creating 100 different `.gif` files, one for each possible width of the bar. I also could have used a one-pixel square `.gif` file with equal ease.

```
    output += "<TD WIDTH=100>"
    chartWidth = Math.round(itemTotal[i])
    if (chartWidth > 0) {
       output += "<IMG SRC='chart.gif' HEIGHT=12 WIDTH=" +
           chartWidth + ">"
    }
    output += "</TD></TR>"
    document.write(output)
    output = ""
}
document.write("</TABLE></FORM></CENTER>")
// end -->
</SCRIPT>
</BODY>
</HTML>
```

dhHelp.htm

The only other code worth noting in this application is in the `dhHelp.htm` document, which appears in the lower-right frame of the window. At the end of this document are two links that call separate JavaScript functions in this document's Head section. The Head functions are as follows:

```
<HEAD>
<TITLE>Decision Helper Help</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function goFirst() {
    parent.entryForms.location = "dh1.htm"
    self.location.hash = "help1"
}
function restart() {
    if (confirm("Erase current decision and start a new one?")) {
        parent.initializeCookies()
        parent.entryForms.location = "dh1.htm"
        self.location.hash = "help1"
    }
}

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

One function merely returns the user to the beginning of the sequences for both the entry screens and the help screen. The second function is a rare instance in which a confirm dialog box makes sense: It is about to erase all entered data. If the user says it's okay to go ahead, the parent window's function for initializing all cookies is called, and the navigation for both the entry and help screens goes back to the beginning.

The links at the bottom of the document (see Figure 55-6) are coded to trigger JavaScript functions (rather than navigate to URLs) and include `onMouseOver` event handlers to provide more information about the link in the statusbar:

```
<A HREF="javascript:goFirst()" onMouseOver="window.status='Go back
to beginning to review data...';return true"">Review This Decision
</A>||<A HREF="javascript:restart()"
onMouseOver="window.status='Erase current data and start over...';return true">
Start a New Decision...  </A>
```

Further Thoughts

If you've managed to follow through with this application's discussions, you will agree that it's quite a JavaScript workout. But this application proves that, without a ton of code, JavaScript provides enough functionality to add a great deal of interactivity and pseudo-intelligence to an otherwise flat HTML document.

As an alternative to using cookies for data storage, I have also implemented a version of the application that uses text boxes defined in a frame defined with a row height of 0. This technique further challenges the synchronization of frames during reloading when a user resizes the browser window or navigates with the Back or Forward browser buttons.

This alternate version is located on the CD-ROM for your own investigation and comparison.

Dynamic HTML also offers some possibilities for this application. The entire program can be presented in a no-frame window, with the navigation, interactive content, and instructions frames incorporated into individual positionable objects. The interactive content area can be treated almost like a slide show, with successive pages flying in from one edge.

Not only is this application instructive for many JavaScript techniques, but it is also fun to play with as a user. Some financial Web sites have adapted it to assist visitors with investment decisions. You can use it to dream about where to go on a dream vacation, or help you decide the most ethical of a few paths confronting you in a personal dilemma. There’s something about putting in data, turning a crank, and watching results (with a bar chart to boot!) magically appear on the screen.