Chapter 53: Application: Calculations and Graphics

In This Chapter

Precached images

Math calculations

CGI-like image assembly

When the scripting world had its first pre-release peeks at JavaScript (while Netscape was still calling the language LiveScript), the notion of creating interactive HTML-based calculators captured the imaginations of many page authors. Somewhere on the World Wide Web, you can find probably every kind of special-purpose calculation normally done by scientific calculators and personal computer programs — leaving only weather-modeling calculations to the supercomputers of the world.

In the search for my calculator gift to the JavaScript universe, I looked around for something more graphical. Numbers, by themselves, are pretty boring; so any way the math could be enlivened was fine by me. Having been an electronics hobbyist since I was a kid, I recalled the color-coding of electronic resistor components. The values of these gizmos aren’t printed in plain numbers anywhere. You have to know the code and the meaning of the location of the colored bands to arrive at the value of each one. I thought that this calculator would be fun to play with, even if you don’t know what a resistor is.

The Calculation

To give you an appreciation for the calculation that goes into determining a resistor’s value, here is the way the system works. Three closely spaced color bands determine the resistance value in ohms. The first (leftmost) band is the tens digit; the second (middle) band is the ones digit. Each color has a number from 0 through 9 assigned to it (black = 0, brown = 1, and so on). Therefore, if the first band is brown and the second band is black, the number you start off with is 10. The third band is a multiplier. Each color determines the power of ten by which you multiply the first digits. For example, the red color corresponds to a multiplier of 10^2, so that 10×10^2 equals 1,000 ohms.

A fourth band, if present, indicates the tolerance of the component — how far, plus or minus, the resistance measurement can fluctuate due to variations in the manufacturing process. Gold means a tolerance of plus-or-minus five percent; silver means plus-or-minus 10 percent; and no band means a 20 percent tolerance. A pinch of extra space typically appears between the main group of three-color bands and the one tolerance band.

User Interface Ideas

The quick-and-dirty, non-graphical approach for a user interface was to use a single frame with four SELECT elements defined as pop-up menus (one for each of the four color bands on a resistor), a button to trigger calculation, and a field to show the numeric results.

How dull.

It occurred to me that if I design the art carefully, I can have JavaScript assemble an updated image of the resistor consisting of different slices of art: static images for the resistor's left and right ends, and variable slivers of color bands for the middle. Rather than use the brute force method of creating an image for every possible combination of colors (3,600 images total!), a far more efficient approach is to have one image file for each color (12 colors plus one empty) and enable JavaScript to call them from the server, as needed, in the proper order. If not for client-side JavaScript, a CGI script on the server would have to handle this kind of intelligence and user interaction. But with this system, any dumb server can dish up the image files as called by the JavaScript script.

The first generation of this resistor calculator used two frames, primarily because I needed a second frame to update the calculator's art dynamically while keeping the pop-up color menus stationary. Images couldn't be swapped back in those frontier days, so the lower frame had to be redrawn for each color choice. Fortunately, NN3 and IE4 enabled me to update individual image objects in a loaded document without any document reloading. Moreover, with all the images precached in memory, page users experience no (or virtually no) delay in making a change from one value to another.

The design for the new version is a simple, single-document interface (see Figure 53-1). Four pop-up menus let you match colors of a resistor, whereas the `onChange` event handler in each menu automatically triggers an image and calculation update. To hold the art together on the page, a table border surrounds the images on the page, whereas the numeric value of the component appears in a text field.

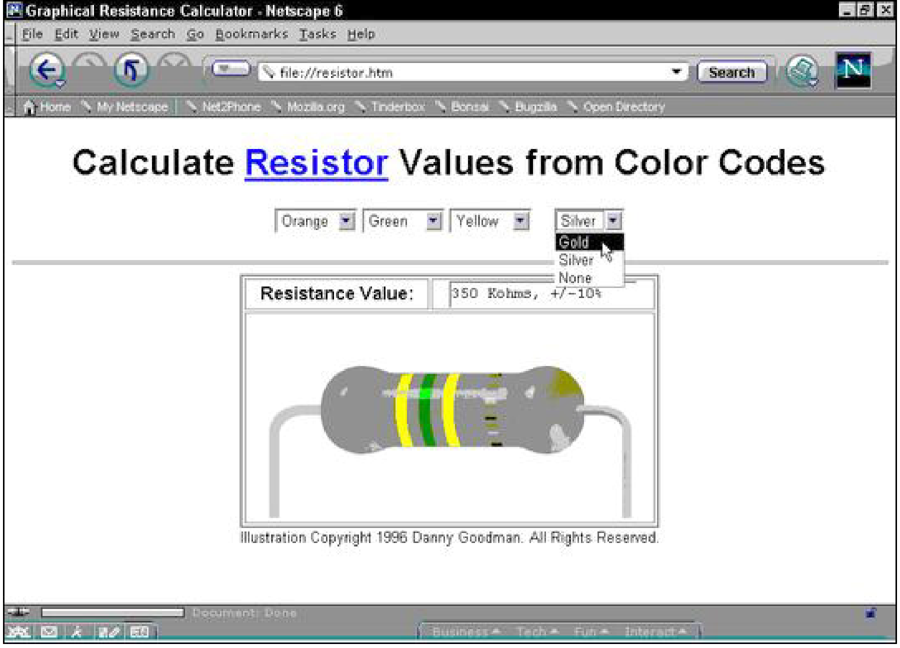
Graphical Resistance Calculator - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
file://resistor.htm
Search
Home My Netscape Net2Phone Mozilla.org Tinderbox Bonsai Bugzilla Open Directory
Calculate Resistor Values from Color Codes
Orange
Green
Yellow
Silver
Gold
Silver
None
Resistance Value:
350 Kohms, +/-10%
Illustration Copyright 1996 Danny Goodman. All Rights Reserved.
Document: Done
Business Tech Fun Interact

Figure 53-1
The Resistor Calculator with images inside a table border

The Code

All the action takes place in the file named `resistor.htm`. A second document is an introductory HTML text document that explains what a resistor is and why you need a calculator to determine a component's value. The article, called *The Path of Least Resistance*, can be viewed in a secondary window from a link in the main window. Here you will be looking only at `resistor.htm`, which has been updated to include style sheets.

The document begins in the traditional way. It specifies a JavaScript 1.1-level language because you will be using several features of that language version:

```
<HTML>
<HEAD>
<TITLE>Graphical Resistance Calculator</TITLE>
<STYLE TYPE="text/css">
BODY {font-family:Arial, Helvetica, serif}
</STYLE>
<SCRIPT LANGUAGE="JavaScript1.1">
<!-- hide script from nonscriptable browsers
```

Basic arrays

In calculating the resistance, the script needs to know the multiplier value for each color. If not for the last two multiplier values actually being negative multipliers (for example, 10^{-1} and 10^{-2}), I could have used the index values without having to create this array. But the two out-of-sequence values at the end make it easier to work with an array rather than to try special-casing these instances in later calculations:

```
// create array listing all the multiplier values
var multiplier = new Array()
multiplier[0] = 0
multiplier[1] = 1
multiplier[2] = 2
multiplier[3] = 3
multiplier[4] = 4
multiplier[5] = 5
multiplier[6] = 6
multiplier[7] = 7
multiplier[8] = 8
multiplier[9] = 9
multiplier[10] = -1
multiplier[11] = -2
// create object listing all tolerance values
var tolerance = new Array()
tolerance[0] = "+/-5%"
tolerance[1] = "+/-10%"
tolerance[2] = "+/-20%"
```

Although the script doesn't do any calculations with the tolerance percentages, it needs to have the strings corresponding to each color for display in the pop-up menu. The `tolerance` array is there for that purpose.

Calculations and formatting

Before the script displays the resistance value, it needs to format the numbers in values that are meaningful to those who know about these values. Just as measures of computer storage bytes, high quantities of ohms are preceded with "kilo" and "meg" prefixes, commonly abbreviated with the "K" and "M" letters. The `format()` function determines the order of magnitude of the final calculation (from another function shown in the following section) and formats the results with the proper unit of measure:

```
// format large values into kilo and meg
function format(ohmage) {
    if (ohmage >= 1e6) {
        ohmage /= 1e6
        return "" + ohmage + " Mohms"
    } else {
        if (ohmage >= 1e3) {
            ohmage /= 1e3
            return "" + ohmage + " Kohms"
        } else {
            return "" + ohmage + " ohms"
        }
    }
}
```

The selections from the pop-up menus meet the calculation formulas of resistors in the `calcOhms()` function. Because this function is triggered indirectly by each of the SELECT objects, sending any of their parameters to the function is a waste of effort. Moreover, the `calcOhms()` function is invoked by the `onLoad` event handler, which is not tied to the form or its controls. Therefore, the function obtains the reference to the form and then extracts the necessary values of the four SELECT objects by using explicit (named) references. Each value is stored in a local variable for convenience in completing the ensuing calculation.

Recalling the rules used to calculate values of the resistor bands, the first statement of the calculation multiplies the "tens" pop-up value times 10 to determine the tens digit and then adds the ones digit. From there, the combined value is multiplied by the exponent value of the selected multiplier value. Notice that the expression first assembles the value as a string to concatenate the exponent factor and then evaluates it to a number. Although I try to avoid the `eval()` function because of its slow performance, the one call per calculation is not a performance issue at all. The evaluated number is passed to the `format()` function for proper formatting (and setting of order of magnitude). In the meantime, the tolerance value is extracted from its array, and the combined string is plugged into the result text field (which is in a separate form, as described later):

```
// calculate resistance and tolerance values
function calcOhms() {
    var form = document.forms["rescalc"]
    var d1 = form.tensSelect.selectedIndex
    var d2 = form.onesSelect.selectedIndex
    var m = form.multiplierSelect.selectedIndex
    var t = form.toleranceSelect.selectedIndex
    var ohmage = (d1 * 10) + d2
    ohmage = eval("" + ohmage + "e" + multiplier[m])
    ohmage = format(ohmage)
    var tol = tolerance[t]
    document.forms["ouput"].result.value = ohmage + ", " + tol
}
```

Preloading images

As part of the script that runs when the document loads, the next group of statements precaches all possible images that can be displayed for the resistor art. For added scripting convenience, the color names are loaded into an array. With the help of that just-created array of color names, you then create another array (`imageDB`), which both generates `Image` objects for each image file and assigns a URL to each image. Notice an important subtlety about the index values being used to create each entry of the `imageDB` array: Each index is a `colorArray` entry, which is the name of the color. As you found out in Chapter 37, if you create an array element with a named index, you must use that style of index to retrieve the data thereafter; you cannot switch arbitrarily between numeric indexes and names. As you see in a moment, this named index provides a critical link between the choices a user makes in the pop-up lists and the image objects being updated with the proper image file.

```
// pre-load all color images into image cache
var colorArray = new Array("Black","Blue","Brown","Gold","Gray",
    "Green","None","Orange","Red","Silver","Violet","White","Yellow")
var imageDB = new Array()
for (i = 0; i < colorArray.length; i++) {
    imageDB[colorArray[i]] = new Image(21,182)
    imageDB[colorArray[i]].src = colorArray[i] + ".gif"
}
```

The act of assigning a URL to the `src` property of an `Image` object instructs the browser to pre-load the image file into memory. This pre-loading happens as the document is loading, so another few seconds of delay won't be noticed by the user.

Changing images on the fly

The next four functions are invoked by their respective SELECT object's `onChange` event handler. For example, after a user makes a new choice in the first SELECT object (the "tens" value color selector), that SELECT object reference is passed to the `setTens()` function. Its job is to extract the text of the choice and use that text as the index into the `imageDB` array. Alternatively, the color name can also be assigned to the `VALUE` attribute of each OPTION, and the `value` property read here. You need this connection to assign the `src` property of that array entry to the `src` property of the image that you see on the page (defined in the following section). This assignment is what enables the images of the resistor to be updated instantaneously — just the one image "slice" affected by the user's choice in a SELECT object.

```
function setTens(choice) {
    var tensColor = choice.options[choice.selectedIndex].text
    document.tens.src = imageDB[tensColor].src
    calcOhms()
}
function setOnes(choice) {
    var onesColor = choice.options[choice.selectedIndex].text
    document.ones.src = imageDB[onesColor].src
    calcOhms()
}
function setMult(choice) {
    var multColor = choice.options[choice.selectedIndex].text
    document.mult.src = imageDB[multColor].src
    calcOhms()
}
function setTol(choice) {
    var tolColor = choice.options[choice.selectedIndex].text
    document.tol.src = imageDB[tolColor].src
    calcOhms()
}
```

The rest of the script for the Head portion of the document merely provides the statements that open the secondary window to display the introductory document:

```
function showIntro() {
    window.open("resintro.htm","",
        "WIDTH=400,HEIGHT=320,LEFT=100,TOP=100")
}
// end script hiding -->
</SCRIPT>
</HEAD>
```

Creating the SELECT objects

A comparatively lengthy part of the document is consumed with the HTML for the four SELECT objects. Notice, however, that the document contains an `onLoad` event handler in the `<BODY>` tag. This handler calculates the results of the currently selected choices whenever the document loads or reloads. If it weren't for this event handler, you would not see the resistor art when the document first appears. Also, because many browsers maintain their form controls' setting while the page is in history, a return to the page later must display the images previously selected in the form.

```
<BODY onLoad="calcOhms()"><CENTER>
<H1>Calculate <A HREF="javascript:showIntro()" onMouseOver="status='An
introduction to the resistor electronic component...';return true">Resistor</A>
Values from Color Codes</H1>
<FORM NAME="rescalc">
<SELECT NAME="tensSelect" onChange="setTens(this)">
    <OPTION SELECTED> Black
    <OPTION> Brown
    <OPTION> Red
    <OPTION> Orange
    <OPTION> Yellow
    <OPTION> Green
    <OPTION> Blue
    <OPTION> Violet
    <OPTION> Gray
    <OPTION> White
</SELECT>
<SELECT NAME="onesSelect" onChange="setOnes(this)">
    <OPTION SELECTED> Black
    <OPTION> Brown
    <OPTION> Red
    <OPTION> Orange
    <OPTION> Yellow
    <OPTION> Green
    <OPTION> Blue
    <OPTION> Violet
    <OPTION> Gray
    <OPTION> White
</SELECT>
<SELECT NAME="multiplierSelect" onChange="setMult(this)">
<OPTION SELECTED> Black
    <OPTION> Brown
    <OPTION> Red
    <OPTION> Orange
    <OPTION> Yellow
    <OPTION> Green
    <OPTION> Blue
    <OPTION> Violet
    <OPTION> Gray
    <OPTION> White
    <OPTION> Gold
    <OPTION> Silver
</SELECT>&nbsp;&nbsp;&nbsp;&nbsp;
<SELECT NAME="toleranceSelect" onChange="setTol(this)">
    <OPTION SELECTED> Gold
    <OPTION> Silver
    <OPTION> None
</SELECT>
</FORM>
<HR>
```

Drawing the initial images

The balance of the document, mostly in JavaScript, is devoted to creating the table and image objects whose `src` properties will be modified with each choice of a SELECT object. The act of assembling the HTML for the image table occurs right after the SELECT objects have rendered. References to those SELECT elements are required in order to extract the currently selected values. If the FORM element that holds the SELECT elements is not closed, you can't build a valid (and backward compatible) reference to the SELECT elements. Therefore, the page contains two forms: One for the SELECT elements; one for the output text box inside the table.

```
<SCRIPT LANGUAGE="JavaScript1.1">
var form = document.forms["input"]
var tensDigit = form.tensSelect.selectedIndex
var tensColor = form.tensSelect.options[tensDigit].text
var onesDigit = form.onesSelect.selectedIndex
var onesColor = form.onesSelect.options[onesDigit].text
var multDigit = form.multiplierSelect.selectedIndex
var multColor = form.multiplierSelect.options[multDigit].text
var tolDigit = form.toleranceSelect.selectedIndex
var tolColor = form.toleranceSelect.options[tolDigit].text

var table = "<TABLE BORDER=2><FORM NAME='output'>"
table += "<TR><TH ALIGN=middle>Resistance Value:</TH>"
table += "<TH ALIGN='middle'><INPUT TYPE='text' NAME='result' " +
    "SIZE=20 onFocus='this.blur()'>"
table += "</TH></TR><TR><TD COLSPAN=2>"
table += "<IMG SRC='resleft.gif' WIDTH=127 HEIGHT=182>" +
    "<IMG SRC='" + tensColor + ".gif' NAME='tens' WIDTH=21 " +
    "HEIGHT=182><IMG SRC='" + onesColor +
    ".gif' NAME='ones' WIDTH=21 HEIGHT=182>" +
    "<IMG SRC='" + multColor + ".gif' NAME='mult' WIDTH=21 "+
    "HEIGHT=182><IMG SRC='spacer.gif' WIDTH=17 HEIGHT=182>"+
    "<IMG SRC='" + tolColor + ".gif' NAME='tol' WIDTH=21 " +
    "HEIGHT=182><IMG SRC='resright.gif' WIDTH=127 HEIGHT=182>"
table += "</TD></TR></FORM></TABLE>"
document.write(table)
</SCRIPT>
<FONT SIZE=2>Illustration Copyright 1996 Danny Goodman. All Rights
Reserved.</FONT></CENTER>
</BODY>
</HTML>
```

As you can see, inside the images appear in one table cell (in the second row) that contains all seven image objects smashed against each other. To keep the images flush against each other, there can be no spaces or carriage returns between `<IMG>` tags.

Further Thoughts

I am very pleased with the improvements to performance and perceived quality that swappable images and image precaching bring to the current version of this calculator. Images change crisply. Network latency is no longer an issue.

In the layout department, however, annoying differences still exist among different platforms. At one point in the design process, I considered trying to align the pop-up menus with images of the resistor (or callout line images), but the differences in platform rendering of pop-up menus made that idea impractical. At best, I now separate the three left SELECT objects from the right one by way of hard-coded spaces (`&nbsp;`).

You should notice from this exercise that I look for ways to blend JavaScript object data structures with my own data's structure. For example, the SELECT objects serve multiple duties in these scripts. Not only does the text of each option point to an image file of the same name, but the index values of the same options are applied to the calculations. Things don't always work out that nicely, but whenever your scripts bring together user interface elements and data elements, look for algorithmic connections involving names or index integers that you can leverage to create elegant, concise code.