Chapter 51: Application: A “Poor Man’s” Order Form

In This Chapter

Live math on table rows and columns

Number formatting

Code reusability

I hesitate to call the application described in this chapter an “order form” because it is not in any way intended for use as a client-side shopping cart or some of the more advanced e-commerce applications you see on the Web. No, the goal here is to demonstrate how JavaScript can be used to assist users with column-and-row arithmetic, very much like the kinds of arithmetic needed to calculate the total for an order of goods.

While this order form is not linked to any particular online catalog, some or all of it can be used as a piece for a small e-commerce site. The form in the example here requires that users input product descriptions and prices, but there is no reason that a client-side JavaScript shopping cart can’t accumulate the shopper’s choices from catalog pages, and then present them in an order form with product descriptions and prices hard-wired into the table. There still are entry boxes for quantity and selecting local sales tax rates. But all the arithmetic products and sums are calculated quickly on the client with JavaScript.

Along the way, you should also discover how to design code — more specifically, JavaScript data structures — in such a way that they are easily editable by non-scripters who are responsible for updating the embedded data. Therefore, even if you prefer to leave professional e-commerce order processing to server CGIs, you may still pick up a scripting tip or two from this “poor man’s” version of an order form.

Defining the Task

I doubt that any two order forms on the Web are executed precisely the same way. Much of the difference has to do with the way a CGI program on the server wants to receive the data on its way to an order-entry system or database. The rest has to do with how clever the HTML programmer is. To come up with a generalized demonstration, I had to select a methodology and stay with it.

Because the intended goal of this demonstration is to focus on the rows and columns of an order form, I omit the usual name-and-address input elements. Instead, the code deals exclusively with the tabular part of the form, including the footer “stuff” of a form for subtotals, sales tax, shipping, and the grand total.

Another goal is to design the order form with an eye to as much reusability as possible. In other words, I may design the form for one page, but I also want to adapt it to another order form quickly without having to muck around too deeply in complicated HTML and JavaScript code. One giant annoyance that this approach eliminates is the normal HTML repetition of row after row of tags for input fields and table cells. JavaScript can certainly help you out there.

The order form code also demonstrates how to perform math and display results in two decimal places, use the `String.split()` method to make it easy to build arrays of data from comma-delimited lists, and enable JavaScript arrays to handle tons of repetitive work.

The Form Design

Figure 51-1 shows a rather simplified version of an order form as provided in the listings. Many elements of the form are readily adjustable by changing only a few characters near the top of the JavaScript listing. At the end of the chapter, I provide several suggestions for improving the user experience of a form, such as this one.

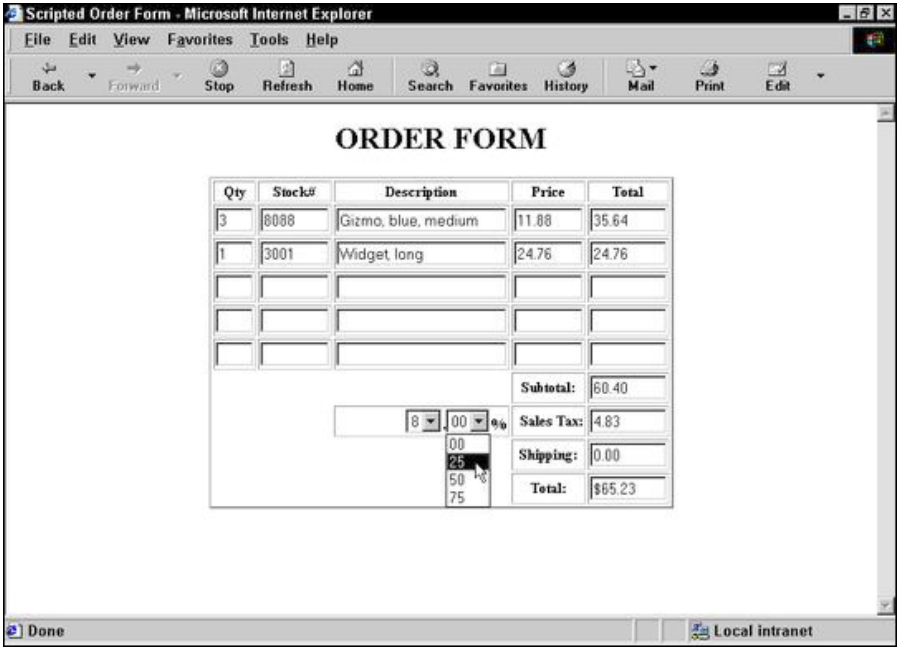

Figure 51-1
The order form display

Because this form is generated as the document loads, JavaScript writes most of the document to reflect the variable choices made in the reusable parts of the script. In fact, in this example, only the document heading is hard-wired in HTML.

The script uses a few JavaScript facilities that aren't available in the earliest browsers, so you have to guard against browsers of other levels reaching this page and receiving script errors when `document.write()` statements fail to find functions defined inside JavaScript 1.1 language script tags. As part of this defense, I defined a JavaScript 1.0 function, called `initialize()`, ahead of any other script. This function is called later in the Body. Because both types of browsers can invoke this function, the Head portion of this document contains an `initialize()` function in both JavaScript 1.0 and JavaScript 1.1 script tags. For JavaScript 1.0 browsers, the function displays a message alerting the user that this form requires a more recent browser. Your message could be more helpful and perhaps even provide a link to another version of the order form. In the JavaScript 1.1 portion, the `initialize()` function is empty, sitting ready to catch and ignore the call made by the document:

```
<HTML>
<HEAD>
<TITLE>Scripted Order Form</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
// displays notice for non-JavaScript 1.1 browsers
function initialize() {
    document.write("This page a more recent browser version.")
}
// -->
</SCRIPT>
```

Global adjustments

The next section is the start of the JavaScript 1.1-level statements and functions that do most of the work for this document. The script begins by initializing three very important global variables. This location is where the author defining the details for the order form also enters information about the column headings, column widths, and number of data entry rows.

```
<SCRIPT LANGUAGE="JavaScript1.1">
<!--
// ** BEGIN GLOBAL ADJUSTMENTS ** //
// Order form columns and rows specifications
// **Column titles CANNOT CONTAIN PERIODS
var columnHeads = "Qty,Stock#,Description,Price,Total".split(",")
var columnWidths = "3,7,20,7,8".split(",")
var numberOfRows = 5
```

The first two assignment statements perform double duty. Not only do they provide the location for customized settings to be entered by the HTML author, but they use the *`string`*`.split()` method to literally create arrays out of their series of comma-delimited strings. At first, this may seem to be a roundabout way to generate an array, because you can also create the array directly with:

```
var columnHeads = new Array("Qty","Stock",...)
```

But the way shown here minimizes the possibility of goofing up the quotes and commas when modifying the data, especially if modification might be attempted by a nonscripter.

So much of the repetitive work to come in this application is built around arrays that it will prove to be extraordinarily convenient to have the column title names and column widths in parallel arrays. The number-of-rows value also plays a role in not only drawing the form, but calculating it as well.

Notice the caveat about periods in column heading strings. You will soon see that these column names are assigned as text object names, which, in turn, are used to build object references to text boxes. Object names cannot have periods in them, so for these column headings to perform their jobs, you have to leave periods out of their names.

As part of the global adjustment area, the `extendRow()` method requires knowledge about which columns are to be multiplied to reach a total for any row:

```
// data entry row math
function extendRow(form,rowNum) {
    // **change 'Qty' and 'Price' to match your column names
    var rowSum = form.Qty[rowNum].value * form.Price[rowNum].value
    // **change 'Total' to match your corresponding column name
    form.Total[rowNum].value = formatNum(rowSum,2)
}
```

This example uses the `Qty`, `Price`, and `Total` fields for math calculations. Those field names are inserted into the references within this function. To calculate the total for each row, the function receives the form object reference and the row number as parameters. As described later, the order form is generated as a kind of array. Each field in a column intentionally has the same name. This scheme enables scripts to access a given field in that column by row number when using the row number as an index to the array of objects bearing the same name. For example, for the first row (row 0), you calculate the total by multiplying the quantity field of row 0 (`form.Qty[0].value`) times the price field of row 0 (`form.Price[0].value`). You then format that value to two places to the right of the decimal and plug that number into the value of the total field for row 0 (`form.Total[0].value`).

The final place where you have to worry about customized information is in the function that adds up the total columns. The function must know the name that you assigned to the total column:

```
function addTotals(form) {
    var subTotal = 0
    for (var i = 0; i < numberOfRows; i++) {
        // **change 'Total' in both spots to match your column name
        subTotal += (form.Total[i].value != "") ?
            parseFloat(form.Total[i].value) : 0
    }
    form.subtotal.value = formatNum(subTotal,2)
    form.tax.value = formatNum(getTax(form,subTotal),2)
    form.total.value = "$" + formatNum((parseFloat(form.subtotal.value) +
        parseFloat(form.tax.value) + parseFloat(form.shipping.value)),2)
}
// ** END GLOBAL ADJUSTMENTS ** //
```

The `addTotals()` function receives the form reference as a parameter, which it uses to read and write data around the form. The first task is to add up the values of the total fields from each of the data-entry rows. Here you need to be specific about the name you assign to that column. To keep code lines to a minimum, you use a conditional expression inside the `for` loop to make additions to the `subTotal` amount only when a value appears in a row's total field. Because all values from text fields are strings, you use `parseFloat()` to convert the values to floating-point numbers before adding them to the `subTotal` variable.

Three more assignment statements fill in the subtotal, tax, and total fields. The subtotal is nothing more than a formatted version of the amount reached at the end of the `for` loop. The task of calculating the sales tax is passed off to another function (described in a following section), but its value is also formatted before being plugged into the sales tax field. For the grand total, you add floating-point-converted values of the subtotal, tax, and shipping fields before slapping a dollar sign in front of the result. Even though the three fields contain values formatted to two decimal places, any subsequent math on such floating-point values incurs the minuscule errors that send formatting out to sixteen decimal places. Thus, you must reformat the results after the addition.

Do the math

As you can see from Figure 51-1, the user interface for entering the sales tax is a pair of SELECT elements. This type of interface minimizes the possibility of users entering the value in all kinds of weird formats that, in some cases, would be impossible to parse. The function that calculates the sales tax of the subtotal looks to these select objects for their current settings.

```
function getTax(form,amt){
    var chosenPercent = form.percent[form.percent.selectedIndex].value
    var chosenFraction = form.fraction[form.fraction.selectedIndex].value
    var rate = parseFloat(chosenPercent + "." + chosenFraction) / 100
    return amt * rate
}
```

After receiving the form object reference and subtotal amount as parameters, the function reads the two values chosen in the SELECT elements. The string `value` properties of the SELECT objects are temporarily stored in local variables. To arrive at the actual rate, you concatenate the two portions of the string (joined by an artificial decimal point) and `parseFloat()` the string to get a number that you can then divide by 100. The product of the subtotal times the rate is returned to the calling statement (in the preceding `addTotals()` function).

All of the calculation that ripples through the order form is controlled by a single `calculate()` function:

```
function calculate(form,rowNum) {
    extendRow(form,rowNum)
    addTotals(form)
}
```

This function is called by any object that affects the total of any row. Such a request includes both the form object reference and the row number. This information lets the single affected row, and then the totals column, be recalculated. Changes to some objects, such as the sales tax SELECT objects, affect only the totals column, so they will call `addTotals()` function directly rather than this function (the rows don't need recalculation).

Number formatting, as explained in Chapter 35, is a detail that scripters must handle themselves (unless you are designing for IE5.5+ and NN6+, which include the *`number`*`.toFixed()` method for number formatting). We can borrow the formatting code from Chapter 35, and use it here as-is:

```
function formatNum(expr,decplaces) {
    var str = (Math.round(parseFloat(expr) *
        Math.pow(10,decplaces))).toString()
    while (str.length <= decplaces) {
        str = "0" + str
   }
    var decpoint = str.length - decplaces
    return str.substring(0,decpoint) + "." +
    str.substring(decpoint,str.length)
}
```

Being able to pick up this function from a different application should reinforce the advantage to writing functions to be as generalizable as possible. Rather than building page-specific references into the formatting function, it accepts parameters that could come from anywhere. Page specifics are left to another function that deals with reading and writing text box values.

Cooking up some HTML

As we near the end of the scripting part of the document's Head section, we come to two functions that are invoked later to assemble some table-oriented HTML based on the global settings made at the top. One function assembles the row of the table that contains the column headings:

```
function makeTitleRow() {
    var titleRow = "<TR>"
    for (var i = 0; i < columnHeads.length; i++) {
        titleRow += "<TH>" + columnHeads[i] + "</TH>"
    }
    titleRow += "</TR>"
    return titleRow
}
```

The heart of the `makeTitleRow()` function is the `for` loop, which makes simple `<TH>` tags out of the text entries in the `columnHeads` array defined earlier. All this function does is assemble the HTML. A `document.write()` method in the Body puts this HTML into the document.

```
function makeOneRow(rowNum) {
    var oneRow = "<TR>"
    for (var i = 0; i < columnHeads.length; i++) {
        oneRow += "<TD ALIGN=middle><INPUT TYPE=text SIZE=" +
            columnWidths[i] + " NAME=\'" + columnHeads[i] +
            "\' onChange='calculate(this.form," + rowNum + ")'></TD>"
```

```
    }
    oneRow += "</TR>"
    return oneRow
}
```

Creating a row of entry fields is a bit more complex, but not much. Instead of assigning just a word to each cell, you assemble an entire `<INPUT>` object definition. You use the `columnWidths` array to define the size for each field (which therefore defines the width of the table cell in the column). `columnHead` values are assigned to the field's NAME attribute. Each column's fields have the same name, no matter how many rows exist. Finally, the `onChange` event handler invokes the `calculate()` method, passing the form and, most importantly, the row number, which comes into this function as a parameter (see the following section).

Some JavaScript language cleanup

The final function in the Head script is an empty function for `initialize()`. This function is the one that JavaScript 1.1-level browsers activate after the document loads into them:

```
// do nothing when JavaScript 1.1 browser calls here
function initialize() {}
//-->
</SCRIPT>
</HEAD>
<BODY>
<CENTER>
<H1>ORDER FORM</H1>
<FORM>
<TABLE BORDER=2>
<SCRIPT LANGUAGE="JavaScript">
<!--
initialize()
// -->
</SCRIPT>
```

From there, you start the `<BODY>` definition, including a simple header. You immediately go into the form and table definitions. A JavaScript script that will be run by all versions of JavaScript invokes the `initialize()` function. JavaScript 1.0-level browsers execute the `initialize()` function in the topmost version in the Head so that they display the warning message in the document's body; JavaScript 1.1-level browsers execute the empty function you see.

Tedium lost

Believe it or not, all of the rows of data-entry fields in the table are defined by the handful of JavaScript statements that follow:

```
<SCRIPT LANGUAGE="JavaScript1.1">
document.write(makeTitleRow())
// order form entry rows
for (var i = 0; i < numberOfRows; i++) {
    document.write(makeOneRow(i))
}
```

The first function to be called is the `makeTitleRow()` function, which returns the HTML for the table's column headings. Then a very simple `for` loop writes as many rows of the field cells as defined in the global value near the top of the document. Notice how the index of the loop, which corresponds to the row number, is passed to the `makeOneRow()` function, so that it can assign that row number to its relevant statements. Therefore, these few statements generate as many entry rows as you need.

Tedium regained

What follows in the script writes the rest of the form to the screen. To make these fields as intelligent as possible, the scripts must take the number of columns into consideration. A number of empty-space cells must also be defined (again, calculated according to the number of columns). Finally, the code-consuming SELECT element definitions must also be in this segment of the code.

```
// order form footer stuff (subtotal, sales tax, shipping, total)
var colSpacer = "<TR><TD COLSPAN=" +
    (columnWidths.length - 2) + "></TD>"
document.write(colSpacer)
document.write("<TH>Subtotal:</TH>")
document.write("<TD><INPUT TYPE=text SIZE=" +
    columnWidths[columnWidths.length - 1] + " NAME=subtotal></TR>")
document.write("<TR><TD COLSPAN=" +
    (columnWidths.length - 3) + "></TD>")
var tax1 = "<SELECT NAME=percent
onChange='addTotals(this.form)'><OPTION>0<OPTION>1<OPTION>2<OPTION>3"
tax1 += "<OPTION VALUE=1>1<OPTION VALUE=2>2<OPTION VALUE=3>3"
tax1 += "<OPTION VALUE=4>4<OPTION VALUE=5>5<OPTION VALUE=6>6"
tax1 += "<OPTION VALUE=7>7<OPTION VALUE=8>8<OPTION VALUE=9>9"
tax1 += "</SELECT>"
var tax2 = "<SELECT NAME=fraction onChange='addTotals(this.form)'>"
tax2 += "<OPTION VALUE=0>00<OPTION VALUE=25>25"
tax2 += "<OPTION VALUE=5>50<OPTION VALUE=75>75</SELECT>"
document.write("<TH ALIGN=RIGHT>" + tax1 + "." + tax2 + "\%</TH>")
document.write("<TH ALIGN=RIGHT>Sales Tax:</TH>")
document.write("<TD><INPUT TYPE=text SIZE=" +
    columnWidths[columnWidths.length - 1] + " NAME=tax VALUE=0.00></TR>")
document.write(colSpacer)
document.write("<TH>Shipping:</TH>")
document.write("<TD><INPUT TYPE=text SIZE=" +
    columnWidths[columnWidths.length - 1] + " NAME=shipping VALUE=0.00
onChange='addTotals(this.form)'></TR>")
document.write(colSpacer)
document.write("<TH>Total:</TH>")
document.write("<TD><INPUT TYPE=text SIZE=" +
    columnWidths[columnWidths.length - 1] + " NAME=total></TR>")
</SCRIPT>

</TABLE></FORM>
</BODY>
</HTML>
```

To gain a better understanding of how the script assembles the HTML for this part of the table, start by looking at the `colSpacer` variable. This variable contains a table cell definition that must span all but the rightmost two columns. Thus, the `COLSPAN` attribute is calculated based on the length of the `columnWidths` array (minus two for the columns we need for data). Therefore, to write the line for the subtotal field, you start by writing one of these column spacers, followed by the `<TH>` type of cell with the label in

it. For the actual field, you must size it to match the fields for the rest of the column. That's why you summon the value of the last `columnWidths` value for the `SIZE` attribute. You use similar machinations for the Shipping and Total lines of the form footer material.

In between these locations, you define the Sales Tax SELECT objects (and a column spacer that is one cell narrower than the other one you used). To reduce the risk of data-entry error and to allow for a wide variety of values without needing a 40-item pop-up list, I divided the choices into two components and then display the decimal point and percentage symbol in hard copy. Both SELECT objects trigger the `addTotals()` function to recalculate the rightmost column of the form.

Sometimes, it seems odd that you can script four lines of code to get 20 rows of a table, yet it takes twenty lines of code to get only four more complex rows of a table. Such are the incongruities of the JavaScripter's life.

Further Thoughts

Depending on the catalog of products or services being sold through this order form, the first improvement I would make is to automate the entry of stock number and description. For example, if the list of all product numbers isn't that large, you may want to consider dropping a SELECT element into each cell of the Description column. Then, after a user makes a selection, the `onChange` event handler performs a lookup through a product array and automatically plugs in the description and unit price. In any version of this form, you also need to perform data validation for crucial calculation fields, such as quantity.

In a CGI-based system that receives data from this form, individual fields do not have unique names, as mentioned earlier. All `Qty` fields, for instance, have that name. But when the form is submitted, the name-value pairs appear in a fixed order every time. Your CGI program can pull the data apart partly by field name, partly by position. The same goes for a program you may build to extract form data that is e-mailed to you rather than sent as a CGI request.

Some of the other online order forms I've seen include reset buttons for every row or a column of checkmarks that lets users select one or more rows for deletion or resetting. Remember that people make mistakes and change their minds while ordering online. Give them plenty of opportunity to recover easily. If getting out of jam is too much trouble, they will head for the History list or Back button, and that valued order will be, well, history.