Chapter 57: Application: Transforming XML Data Islands

In This Chapter

Designing XML data islands

Complex JavaScript data structures

Advanced array sorting

Dynamic tables

Chapter 52 ends with an example of an interactive outliner whose data arrives in XML format. The data is embedded in an HTML document inside an XML data island, which is thus far supported only on the Windows versions of IE5 and later. The application described in this chapter picks up from there.

As you recall from the Chapter 52 outline, the node structure of the XML data was used as a guide to the structure for a one-time rendering of HTML elements. There was a one-to-one correlation between XML element nesting and the HTML element nesting. Adjusting style sheet properties for displaying or hiding elements controlled all interactivity. What you're about to see here is a case for converting XML into JavaScript objects that can be used multiple times as a convenient data source for HTML that is displayed in any number of formats. In particular, you see how JavaScript's array sorting prowess supplies XML-supplied data with extraordinary flexibility in presentation.

You will see a lot of code in this chapter. The code is presented here as a way to demonstrate the potential for rich data handling. At the same time, the code may provide ideas for server-side processing of XML data being output to the client. If a server program can convert the XML data into the shortcut object and array notation of Version 4 browsers or later, suddenly a broader range of browsers is capable of dealing with data stored as XML on the server.

Application Overview

Understanding the data is a good place to start in describing this application. The scenario is a small American company (despite its grandiose name: GiantCo) that has divided the country into three sales regions. Two of the regions have two sales representatives, while the third region has three reps. The time is at the end of a fiscal year, at which point the management wants to review and present the performance of each salesperson. An XML report delivers the sales forecast and actual sales per quarter for each sales rep. A single HTML and JavaScript page (with the XML data embedded as a data island inside an IE

<XML> tag) is charged with not only displaying the raw tabular data, but also allowing for a variety of views and sorting possibilities so that management can analyze performance by sales rep and region, as well as by quarter.

A server-based searching and reporting program collects the requested data and outputs each sales rep's record in an XML structure, such as the following one:

```
<SALESREP>
    <EMPLOYEEID>12345</EMPLOYEEID>
    <CONTACTINFO>
        <FIRSTNAME>Brenda</FIRSTNAME>
        <LASTNAME>Smith</LASTNAME>
        <EMAIL>brendas@giantco.com</EMAIL>
        <PHONE>312-555-9923</PHONE>
        <FAX>312-555-9901</FAX>
    </CONTACTINFO>
    <MANAGER>
        <EMPLOYEEID>02934</EMPLOYEEID>
        <FIRSTNAME>Alistair</FIRSTNAME>
        <LASTNAME>Renfield</LASTNAME>
    </MANAGER>
    <REGION>Central</REGION>
    <SALESRECORD>
        <PERIOD>
            <ID>Q1_2000</ID>
            <FORECAST>300000</FORECAST>
            <ACTUAL>316050</ACTUAL>
        </PERIOD>
        <PERIOD>
            <ID>Q2_2000</ID>
            <FORECAST>280000</FORECAST>
            <ACTUAL>285922</ACTUAL>
        </PERIOD>
        <PERIOD>
            <ID>Q3_2000</ID>
            <FORECAST>423000</FORECAST>
            <ACTUAL>432930</ACTUAL>
        </PERIOD>
        <PERIOD>
            <ID>Q4_2000</ID>
            <FORECAST>390000</FORECAST>
            <ACTUAL>399200</ACTUAL>
        </PERIOD>
    </SALESRECORD>
</SALESREP>
```

As you can see, the data consists of several larger blocks, such as contact information, a pointer to the rep's manager, and then the details of each quarterly period's forecast and actual sales. The goal is to present the data in table form with a structure similarly shown in Figure 57-1. Not only is the raw data presented, but numerous calculations are also made on the results, such as the percentage of quota attained for each reporting period, plus totals along each axis of the spreadsheet-like table.

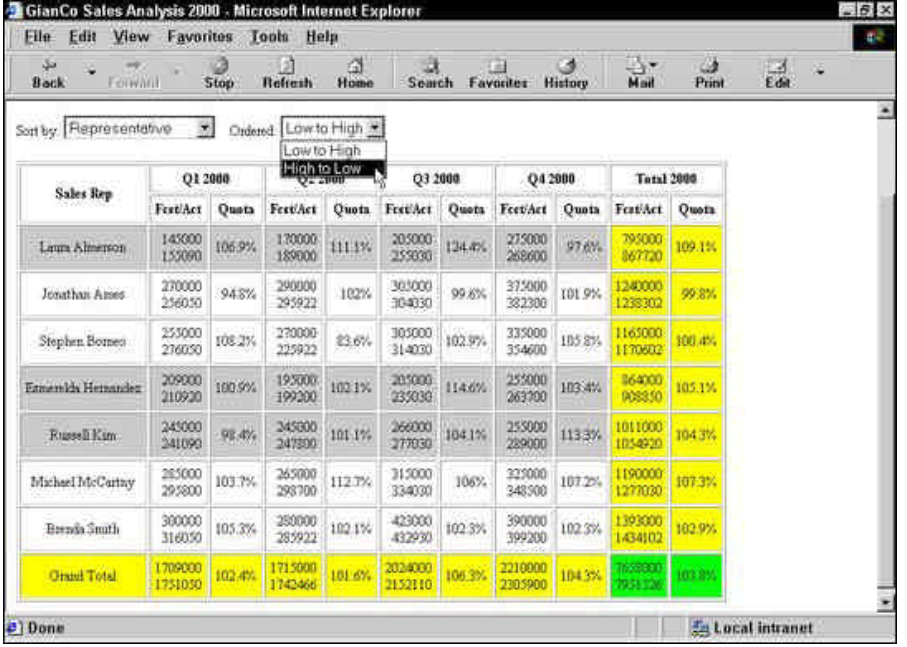

Figure 57-1

One view of the XML data output

Just above the table are two SELECT elements. These controls' labels indicate that the table's data can be sorted by a number of criteria and the results of each sort can be ordered in different ways. Sorting in the example offers the following possibilities:

```
Representative's Name
Sales Region
Q1 Forecast
Q1 Actual
Q1 Performance
[the last three also for Q2, Q3, Q4]
Total Forecast
Total Actual
Total Performance
```

Ordering of the sorted results is a choice between "Low to High" or "High to Low." While ordering of most sort categories is obviously based on numeric value, the sorting of the representatives' names is based on the alphabetical order of the last names. One other point about the user interface is that the design needs to signify via table cell background color the sales region of each representative. The colors aren't easily distinguishable in Figure 57-1, but if you open the actual example listing in IE5+/Windows on your computer, you will see the coloration.

Implementation Plan

Clearly all the data needed for numerous sorted and ordered views arrives in one batch in the XML island. Despite the element and node referencing properties and methods of the W3C DOM, trying to use the XML elements as the sole data store for scripts to sort the data each time would be impractical. For one thing, none of the elements have ID

attributes — there's no need for it in the XML stored on the server database. And even if they did have IDs, how would scripts that you desire to write for generalizability make use of them unless the IDs were generated in a well-known sequence? Moreover, after a sales rep's record is rendered in the table, how easy would it be to dive back into that record and drill down for further information, such as the name of a representative's manager?

A solution that can empower the page author in this case is to use the node-walking properties and methods of the W3C DOM to assemble a JavaScript-structured database while the page loads. In other words, the conversion is performed just once during page loading, and the JavaScript version is preserved in an array (of XML "records" in this case) as a global variable. Any transformations on the data can be done from the JavaScript database with the help of additional powers of the language.

Given that route, the basic operation of the scripting of the page is schematically simple:

1. Convert the XML into an array of objects at load time.
2. Predefine all necessary sorting functions based on properties of those objects.
3. Provide a function that rebuilds the HTML table each time data is sorted.

With this sequence in mind, now look into the code that does the job.

The Code

Rather than work through the long document in source code order, the following descriptions follow a more functional order. You can open the actual source code file to see where the various functions are positioned. To best understand this application, seeing the "how" rather than the "where" is more important. Also, many of the code lines (even some single expressions) are too wide for the printed page and therefore break unnaturally in the listings that follow. Trust the formatting of the source file on the CD-ROM.

Style sheets

For the example provided on the CD-ROM, one set of style sheet rules is embedded in the HTML document. As you can see from the rule selectors, many are tied to very specific classes of table-related elements used to render the content. In a production version of this application, I would expect that there would be more and quite different views of the data available to the users, such as bar charts for each salesperson or region. Each view would likely require its own unique set of style sheet rules. In such a scenario, the proper implementation would be to use the LINK element to bring in a different external style sheet file for each view type. All could be linked in at the outset, but only the current `styleSheet` object would be enabled.

```
<STYLE TYPE="text/css">
XML {display:none}
TD {text-align:right}
```

```
TD.rep, TD.grandTotalLabel {text-align:center}
TR.East {background-color:#FFFFCC}
TR.Central {background-color:#CCFFFF}
TR.West {background-color:#FFCCCC}
TR.QTotal {background-color:#FFFF00}
TD.repTotal {background-color:#FFFF00}
TD.grandTotal{background-color:#00FF00}
H1 {font-family:"Comic Sans MS",Helvetica,sans-serif}
</STYLE>
```

One style sheet rule is essential: The one that suppresses the rendering of any XML element. That data is hidden from the user's view.

Initialization sequence

An `onLoad` event handler invokes the `init()` function, which sets a lot of machinery in motion to get the document ready for user interaction. Its most important job is running a `for` loop that builds the JavaScript database from the XML elements. Next, it sorts the database based on the current choice in the sorting SELECT element. The sorting function ends by triggering the rendering of the table. These three actions correspond to the fundamental operation of the entire application.

```
// initialize global variable that stores JavaScript data
var db = new Array()

// Initialization called by onLoad
function init() {
    for (var i = 0;
    i <
document.getElementById("reports").getElementsByTagName("SALESREP").length;
    i++) {
        db[db.length] = getOneSalesRep(i)
    }
    selectSort(document.getElementById("sortChooser"))
}
```

Converting the data

The controlling factor for creating the JavaScript database is the structure of the XML data island. As you may recall, the elements inside the XML data island can be accessed only through a reference to the XML container. The ID of that element in this application is `reports`. Data for each sales rep is contained by a SALESREP element. The number of SALESREP elements determines how many records (JavaScript objects) are to be added to the `db` array. A call to the `getOneSalesRep()` function creates an object for each sales representative's data.

Despite the length of the `getOneSalesRep()` function, its operation is very straightforward. Most of the statements do nothing more than retrieve the data inside the various XML elements within a SALESREP container and assign that data to a like-named property of the custom object. Following the structure of the XML example shown earlier in this chapter, you can see where some properties of a JavaScript object representing the data are, themselves, objects or arrays. For example, one of the properties is called `manager`, corresponding to the MANAGER element. But that element has nested items inside. Then, making those nested elements properties of a `manager` object

is only natural. Similarly, the repetitive nature of the data within each of the four quarterly periods calls for even greater nesting: The object property named `sales` is an array, with each item of the array corresponding to one of the periods. Each period also has three properties (a period ID, forecast sales, and actual sales). Thus, the `sales` property is an array of objects.

```
function getOneSalesRep(i) {
    // create new, empty object
    var oneRecord = new Object()
    // get a shortcut reference to one SALESREP element
    var oneElem =
document.getElementById("reports").getElementsByTagName("SALESREP")[i]
    // start assigning element data to oneRecord object properties
    oneRecord.id =
oneElem.getElementsByTagName("EMPLOYEEID")[0].firstChild.data
    var contactInfoElem =
oneElem.getElementsByTagName("CONTACTINFO")[0]
    oneRecord.firstName =
contactInfoElem.getElementsByTagName("FIRSTNAME")[0].firstChild.data
    oneRecord.lastName =
contactInfoElem.getElementsByTagName("LASTNAME")[0].firstChild.data
    oneRecord.eMail =
contactInfoElem.getElementsByTagName("EMAIL")[0].firstChild.data
    oneRecord.phone =
contactInfoElem.getElementsByTagName("PHONE")[0].firstChild.data
    oneRecord.fax =
contactInfoElem.getElementsByTagName("FAX")[0].firstChild.data
    // make the manager property its own object
    oneRecord.manager = new Object()

    // get a shortcut reference to the MANAGER element
    var oneMgrElem = oneElem.getElementsByTagName("MANAGER")[0]
    // start assigning element data to manager object properties
    oneRecord.manager.id =
oneMgrElem.getElementsByTagName("EMPLOYEEID")[0].firstChild.data
        oneRecord.manager.firstName =
oneMgrElem.getElementsByTagName("FIRSTNAME")[0].firstChild.data
        oneRecord.manager.lastName =
oneMgrElem.getElementsByTagName("LASTNAME")[0].firstChild.data
    oneRecord.region =
oneElem.getElementsByTagName("REGION")[0].firstChild.data

    // make the sales property a new array
    oneRecord.sales = new Array()
    // get a shortcut reference to the collection of
    // periods in the SALESRECORD element
    var allPeriods =
oneElem.getElementsByTagName("SALESRECORD")[0].childNodes
    var temp
    var accumForecast = 0, accumActual = 0
    // loop through periods
    for (var i = 0; i < allPeriods.length; i++) {
        if (allPeriods[i].nodeType == 1) {
            // make new object for a period's data
            temp = new Object()
            // start assigning period data to the new object
            temp.period =
allPeriods[i].getElementsByTagName("ID")[0].firstChild.data
            temp.forecast =
parseInt(allPeriods[i].getElementsByTagName("FORECAST")[0].firstChild.data)
            temp.actual =
parseInt(allPeriods[i].getElementsByTagName("ACTUAL")[0].firstChild.data)
            // run analysis on two properties and preserve result
            temp.quotaPct = getPercentage(temp.actual, temp.forecast)
            oneRecord.sales[temp.period] = temp
```

```
            // accumulate totals for later
            accumForecast += temp.forecast
            accumActual += temp.actual
        }
    }
    // preserve accumulated totals as oneRecord properties
    oneRecord.totalForecast = accumForecast
    oneRecord.totalActual = accumActual
    // run analysis on accumulated totals
    oneRecord.totalQuotaPct = getPercentage(accumActual, accumForecast)
     // hand back the stuffed object to be put into the db array
    return oneRecord
}
// calculate percentage of actual/forecast
function getPercentage(actual, forecast) {
    var pct = (actual/forecast * 100) + ""
    pct = pct.match(/\d*\.\d/)
    return parseFloat(pct)
}
```

Assuming that the raw XML database stores only the sales forecast and actual dollar figures, it is up to analysis programs to perform their own calculations, such as how the actual sales compare against the forecasts. As you saw in the illustration of the rendered table, this application not only displays the percentage differences between the pairs of values, but it also provides sorting facilities on those percentages. To speed the sorting, the percentages are calculated as the JavaScript database is being accumulated, and the percentages are stored as properties of each object. Percentage calculation is called upon in two different statements of the `getOneSalesRep()` function, so that the calculation is broken out to its own function, `getPercentage()`. In that function, the two passed values are massaged to calculate the percentage value, and then the string result is formatted to no more than one digit to the right of the decimal (by way of a regular expression). The value returned for the property assignment is converted to a number data type, because sorting on these values needs to be done according to numeric sorting, rather than string sorting.

You can already get a glimpse at the contribution JavaScript is making to the scripted representation of the data transmitted in XML form. By virtue of planning for subsequent calculations, the JavaScript object contains considerably more information than was originally delivered, yet all the properties are derived from “hard” data supplied by the server database.

Sorting the JavaScript database

With so many sorting keys for the user to choose from, it’s no surprise that sorting code occupies a good number of script lines in this application. All sorting code consists of two major blocks: *dispatching* and *sorting*.

The dispatching portion is nothing more than one gigantic `switch` construction that sends execution to one of the seventeen (!) sorting functions that match whichever sort key is chosen in the SELECT element on the page. This dispatcher function, `selectSort()`, is also invoked from the `init()` function at load time. Thus, if the user makes a choice in the page, navigates to another page, and then returns with the page

still showing the previous selection, the `onLoad` event handler will reconstruct the table precisely as it was. When sorting is completed, the table is drawn, as you see shortly.

```
// begin sorting routines
function selectSort(chooser) {
    switch (chooser.value) {
        case "byRep" :
            db.sort(sortDBByRep)
            break
        case "byRegion" :
            db.sort(sortDBByRegion)
            break
        case "byQ1Fcst" :
            db.sort(sortDBByQ1Fcst)
            break
        case "byQ1Actual" :
            db.sort(sortDBByQ1Actual)
            break
       case "byQ1Quota" :
            db.sort(sortDBByQ1Quota)
            break
        case "byQ2Fcst" :
            db.sort(sortDBByQ2Fcst)
            break
        case "byQ2Actual" :
            db.sort(sortDBByQ2Actual)
            break
        case "byQ2Quota" :
            db.sort(sortDBByQ2Quota)
            break
        case "byQ3Fcst" :
            db.sort(sortDBByQ3Fcst)
            break
        case "byQ3Actual" :
            db.sort(sortDBByQ3Actual)
            break
        case "byQ3Quota" :
            db.sort(sortDBByQ3Quota)
            break
        case "byQ4Fcst" :
            db.sort(sortDBByQ4Fcst)
            break
        case "byQ4Actual" :
            db.sort(sortDBByQ4Actual)
            break
        case "byQ4Quota" :
            db.sort(sortDBByQ4Quota)
            break
        case "byTotalFcst" :
            db.sort(sortDBByTotalFcst)
            break
        case "byTotalActual" :
            db.sort(sortDBByTotalActual)
            break
        case "byTotalQuota" :
            db.sort(sortDBByTotalQuota)
            break
    }
    drawTextTable()
}
```

Each specific sorting routine is a function that automatically works repeatedly on pairs of entries of an array (see Chapter 37). Array entries here (from the `db` array) are objects — and rather complex objects at that. The benefit of using JavaScript array sorting is that the sorting can be performed on any property of objects stored in the array. For example,

sorting on the `lastName` property of each `db` array object is based on a comparison of the `lastName` property for each of the pairs of array entries passed to the `sortDBByRep()` sort function. But looking down a little further, you can see that the mechanism allows sorting on even more deeply nested properties, such as the `sales.Q1_2000.forecast` property of each array entry. If a property in an object can be referenced, it can be used as a sorting property inside one of these functions.

```
function sortDBByRep(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.lastName < b.lastName) ? -1 : 1
    } else {
        return (a.lastName > b.lastName) ? -1 : 1
    }
}
function sortDBByRegion(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.region < b.region) ? -1 : 1
    } else {
        return (a.region > b.region) ? -1 : 1
    }
}
function sortDBByQ1Fcst(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q1_2000.forecast - b.sales.Q1_2000.forecast)
    } else {
        return (b.sales.Q1_2000.forecast - a.sales.Q1_2000.forecast)
    }
}
function sortDBByQ1Actual(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q1_2000.actual - b.sales.Q1_2000.actual)
    } else {
        return (b.sales.Q1_2000.actual - a.sales.Q1_2000.actual)
    }
}
function sortDBByQ1Quota(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q1_2000.quotaPct - b.sales.Q1_2000.quotaPct)
    } else {
        return (b.sales.Q1_2000.quotaPct - a.sales.Q1_2000.quotaPct)
    }
}
function sortDBByQ2Fcst(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q2_2000.forecast - b.sales.Q2_2000.forecast)
    } else {
        return (b.sales.Q2_2000.forecast - a.sales.Q2_2000.forecast)
    }
}
function sortDBByQ2Actual(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q2_2000.actual - b.sales.Q2_2000.actual)
    } else {
        return (b.sales.Q2_2000.actual - a.sales.Q2_2000.actual)
    }
}
function sortDBByQ2Quota(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q2_2000.quotaPct - b.sales.Q2_2000.quotaPct)
    } else {
        return (b.sales.Q2_2000.quotaPct - a.sales.Q2_2000.quotaPct)
    }
}
function sortDBByQ3Fcst(a, b) {
```

```
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q3_2000.forecast - b.sales.Q3_2000.forecast)
    } else {
        return (b.sales.Q3_2000.forecast - a.sales.Q3_2000.forecast)
    }
}
function sortDBByQ3Actual(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q3_2000.actual - b.sales.Q3_2000.actual)
    } else {
        return (b.sales.Q3_2000.actual - a.sales.Q3_2000.actual)
    }
}
function sortDBByQ3Quota(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q3_2000.quotaPct - b.sales.Q3_2000.quotaPct)
    } else {
        return (b.sales.Q3_2000.quotaPct - a.sales.Q3_2000.quotaPct)
    }
}
function sortDBByQ4Fcst(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q4_2000.forecast - b.sales.Q4_2000.forecast)
    } else {
        return (b.sales.Q4_2000.forecast - a.sales.Q4_2000.forecast)
    }
}
function sortDBByQ4Actual(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q4_2000.actual - b.sales.Q4_2000.actual)
    } else {
        return (b.sales.Q4_2000.actual - a.sales.Q4_2000.actual)
    }
}
function sortDBByQ4Quota(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.sales.Q4_2000.quotaPct - b.sales.Q4_2000.quotaPct)
    } else {
        return (b.sales.Q4_2000.quotaPct - a.sales.Q4_2000.quotaPct)
    }
}
function sortDBByTotalFcst(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.totalForecast - b.totalForecast)
    } else {
        return (b.totalForecast - a.totalForecast)
    }
}
function sortDBByTotalActual(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.totalActual - b.totalActual)
    } else {
        return (b.totalActual - a.totalActual)
    }
}
function sortDBByTotalQuota(a, b) {
    if (document.getElementById("orderChooser").value == "inc") {
        return (a.totalQuotaPct - b.totalQuotaPct)
    } else {
        return (b.totalQuotaPct - a.totalQuotaPct)
    }
}
```

For this application, all sorting functions branch in their execution based on the choice made in the “Ordered” SELECT element on the page. The relative position of the two array elements under test in these simple subtraction comparison statements reverses when

the sort order is from low to high (increasing) and when it is from high to low (decreasing). This kind of array sorting is extremely powerful in JavaScript and probably escapes the attention of most scripters.

Constructing the table

As recommended back in Chapter 27's discussion of TABLE and related elements, it is best to manipulate the structure of a TABLE element by way of the specialized methods for tables, rather than mess with nodes and elements. The `drawTextTable()` function is devoted to employing those methods to create the rendered contents of the table below the headers (which are hard-wired in the document's HTML). Composing an eleven-column table requires a bit of code, and the `drawTextTable()`'s length attests to that fact. You can tell by just glancing at the code, however, that for big chunks of it, there is a comfortable regularity that is aided by the JavaScript object that holds the data.

Additional calculations take place while the table's elements are being added to the TABLE element. Column totals are accumulated during the table assembly (row totals are calculated as the object is generated and preserved as properties of the object). A large `for` loop cycles through each (sorted) row of the `db` array; each row of the `db` array corresponds to a row of the table. Class names are assigned to various rows or cells so that they will pick up the style sheet rules defined earlier in the document. Another subtlety of this version is that the `region` property of a sales rep is assigned to the `title` property of a row. If the user pauses the mouse pointer anywhere in that row, the name of the region pops up briefly.

```
function drawTextTable() {
    var newRow
    var accumQ1F = 0, accumQ1A = 0, accumQ2F = 0, accumQ2A = 0
    var accumQ3F = 0, accumQ3A = 0, accumQ4F = 0, accumQ4A = 0
    deleteRows(document.getElementById("mainTableBody"))
    for (var i = 0; i < db.length; i++) {
        newRow = document.getElementById("mainTableBody").insertRow(i)
        newRow.className = db[i].region
        newRow.title = db[i].region + " Region"
        appendCell(newRow, "rep", db[i].firstName + " " + db[i].lastName)
        appendCell(newRow, "Q1", db[i].sales.Q1_2000.forecast + "<BR>" +
            db[i].sales.Q1_2000.actual)
        appendCell(newRow, "Q1", db[i].sales.Q1_2000.quotaPct + "%")
        appendCell(newRow, "Q2", db[i].sales.Q2_2000.forecast + "<BR>" +
            db[i].sales.Q2_2000.actual)
        appendCell(newRow, "Q2", db[i].sales.Q2_2000.quotaPct + "%")
        appendCell(newRow, "Q3", db[i].sales.Q3_2000.forecast + "<BR>" +
            db[i].sales.Q3_2000.actual)
        appendCell(newRow, "Q3", db[i].sales.Q3_2000.quotaPct + "%")
        appendCell(newRow, "Q4", db[i].sales.Q4_2000.forecast + "<BR>" +
            db[i].sales.Q4_2000.actual)
        appendCell(newRow, "Q4", db[i].sales.Q4_2000.quotaPct + "%")
        accumQ1F += db[i].sales.Q1_2000.forecast
        accumQ1A += db[i].sales.Q1_2000.actual
        accumQ2F += db[i].sales.Q2_2000.forecast
        accumQ2A += db[i].sales.Q2_2000.actual
        accumQ3F += db[i].sales.Q3_2000.forecast
        accumQ3A += db[i].sales.Q3_2000.actual
        accumQ4F += db[i].sales.Q4_2000.forecast
        accumQ4A += db[i].sales.Q4_2000.actual
        appendCell(newRow, "repTotal", db[i].totalForecast + "<BR>" +
```

```
            db[i].totalActual)
        appendCell(newRow, "repTotal", db[i].totalQuotaPct + "%")
    }
    newRow = document.getElementById("mainTableBody").insertRow(i)
    newRow.className = "QTotal"
    newRow.title = "Totals"
    appendCell(newRow, "grandTotalLabel", "Grand Total")
    appendCell(newRow, "Q1", accumQ1F + "<BR>" + accumQ1A)
    appendCell(newRow, "Q1", getPercentage(accumQ1A, accumQ1F) + "%")
    appendCell(newRow, "Q2", accumQ2F + "<BR>" + accumQ2A)
    appendCell(newRow, "Q2", getPercentage(accumQ2A, accumQ2F) + "%")
    appendCell(newRow, "Q3", accumQ3F + "<BR>" + accumQ3A)
    appendCell(newRow, "Q3", getPercentage(accumQ3A, accumQ3F) + "%")
    appendCell(newRow, "Q4", accumQ4F + "<BR>" + accumQ4A)
    appendCell(newRow, "Q4", getPercentage(accumQ4A, accumQ4F) + "%")
    var grandTotalFcst = accumQ1F + accumQ2F + accumQ3F + accumQ4F
    var grandTotalActual = accumQ1A + accumQ2A + accumQ3A + accumQ4A
    appendCell(newRow, "grandTotal", grandTotalFcst + "<BR>" +
grandTotalActual)
    appendCell(newRow, "grandTotal",
        getPercentage(grandTotalActual, grandTotalFcst) +  "%")
}
// insert a cell and its content to a recently added row
function appendCell(Trow, Cclass, txt) {
    var newCell = Trow.insertCell(Trow.cells.length)
    newCell.className = Cclass
    newCell.innerHTML = txt
}
// clear previous table content if there is any
function deleteRows(tbl) {
    while (tbl.rows.length > 0) {
        tbl.deleteRow(0)
    }
}
```

Many standalone statements at the end of the `drawTextTable()` function are devoted exclusively to generating the Grand Total row, in which the accumulated column totals are entered. At the same time, the `getPercentage()` function, described earlier, is invoked several times again to derive the quota percentage for the accumulated grand total values in each quarter as well as the complete year.

SELECT controls

To round out the code listing for this application, the values assigned to the two SELECT elements obviously have a lot to do with the execution of numerous functions in this application. Nothing magic takes place here, but you can see the extent of the detail required in assigning script-meaningful hidden values, and human-meaningful text for both SELECT elements. For example, dividing lines help organize the long sort key list into three logical blocks.

```
<P>Sort by: <SELECT ID="sortChooser" onChange="selectSort(this)">
    <OPTION VALUE="byRep">Representative
    <OPTION VALUE="byRegion">Sales Region
    <OPTION VALUE="">----------------------
    <OPTION VALUE="byQ1Fcst">Q1 Forecast
    <OPTION VALUE="byQ1Actual">Q1 Actual
    <OPTION VALUE="byQ1Quota">Q1 Performance
    <OPTION VALUE="byQ2Fcst">Q2 Forecast
    <OPTION VALUE="byQ2Actual">Q2 Actual
    <OPTION VALUE="byQ2Quota">Q2 Performance
    <OPTION VALUE="byQ3Fcst">Q3 Forecast
```

```
    <OPTION VALUE="byQ3Actual">Q3 Actual
    <OPTION VALUE="byQ3Quota">Q3 Performance
    <OPTION VALUE="byQ4Fcst">Q4 Forecast
    <OPTION VALUE="byQ4Actual">Q4 Actual
    <OPTION VALUE="byQ4Quota">Q4 Performance
    <OPTION VALUE="">--------------------
    <OPTION VALUE="byTotalFcst">Total Forecast
    <OPTION VALUE="byTotalActual">Total Actual
    <OPTION VALUE="byTotalQuota">Total Performance
</SELECT>
  
Ordered: <SELECT ID="orderChooser" onChange="selectOrder()">
    <OPTION VALUE="inc">Low to High
    <OPTION VALUE="dec">High to Low
</SELECT>
</P>
```

Dreams of Other Views

Confining the example to just one type of view — a table of numbers — should help you grasp the important processes taking place. But with the XML data converted to JavaScript objects, you can build many other views of the same data into the same page. For example, a script could completely hide the numeric table, and generate a different one that draws bar charts for each sales representative or each region (see Chapter 55 for a scripted bar chart example). The horizontal axis would be the four quarters, and the vertical axis would be dollars or quota percentages. Clicking a bar opens a small window or layer to reveal more detail from the sales representative's record, such as the name of the person's manager. More SELECT elements can let the user select any combination of subsets of the data in either bar chart or numeric table form to facilitate visual comparisons. You might be even more creative and devise ways of showing the data by way of overlapping positioned elements.

The point is that despite the kinds of rendering opportunities afforded by the XSL Transform mechanism (even if you can get comfortable in the syntax and mental model it presents to authors), JavaScript's detailed access to the DOM offers far more potential. Eventually plenty of content authors will mix the two technologies by embedding JavaScript into XSL style sheets to supplement XSL features.

What About NN6?

Microsoft's XML data islands are not (yet anyway) part of the W3C DOM. As NN6 was being readied for release, there was little imperative to implement this feature in the browser (very few convenience features of the IE4+ DOM were adopted in NN6). And, as mentioned elsewhere, without the XML data islands, combining XML and HTML in the same document is not strictly "legal." Oddly enough, the example in this chapter works in NN6, but it is an accident. For one thing, the tag names in the XML data do not overlap with any HTML tag names. But don't take this to mean you can get away with these kinds of constructions. Even if you can force fit your XML into an HTML document to get it to work, you have no guarantee it will work in subsequent browser versions.

To combine the powers of JavaScript and the W3C DOM to operate on XML data in NN6, we have to keep our eyes on availability of the browser's built-in capabilities for standard XSL Transform facilities. Some of it works even in the earliest releases of the new browser, but what works in NN6 doesn't work (or work well) in IE5+, and vice versa. Veteran scripters, who bear scars from battles with DOM incompatibilities, may choose to delay deployments of such content until there is more unanimity among the latest browsers. Browser incompatibilities are responsible for a massive inflation of object model vocabulary (not to mention the thickness of this book). Perhaps the day will come when the code we write for even complex applications will run cleanly on a broad range of installed browsers on a broad range of devices. Don't give up on the dream.