CHAPTER 7

Advanced Client-Side XSLT Techniques

In the previous chapter, you saw some simple examples showing how to work with Extensible Stylesheet Language Transformations (XSLT) on the client side. I showed you how to apply XSLT transformations to both XHTML and XML documents, and you were able to add content and provide structure to the source document.

This chapter works through additional examples that introduce advanced XSLT techniques such as sorting and filtering. You'll work through the following examples:

- Sorting data within an XML document
- Sorting dynamically with JavaScript
- Adding extension functions to Internet Explorer (IE)
- Working with named templates
- Generating JavaScript

Most of these examples work with IE 6, Netscape 8, and Firefox 1.5 and may also work in earlier versions of these browsers. Some examples are specific to IE 6, and none of the examples work in Opera 8.5. Each example includes inline Cascading Style Sheets (CSS) declarations rather than references to external CSS files. While this is not the recommended approach for working with CSS, it makes the examples a little simpler to follow.

I'll finish the chapter with some tips for working with XSLT. By the end of the chapter, you should have a thorough understanding of XSLT and how you can apply client-side transformations in a web browser. As with the previous chapter, you can download the resources from the Source Code area of the Apress web site (`http://www.apress.com`).

Let's start by learning how to sort an XML document using XSLT.

Sorting Data Within an XML Document

In the first example for this chapter, you'll use XSLT to sort XML content within a web browser. If you didn't know how to use XSLT, you could achieve something similar using server-side code or by writing JavaScript. You may be surprised to find out how easy it is to apply sorting with XSLT.

Using XSLT provides the following benefits:

- *The* `<xsl:sort>` *element allows for different types of sorting*: You can sort on multiple levels and on a range of data types, and you can apply both ascending and descending sorts.
- *XSLT sorting is very flexible*: Any new data added to the XML document will be included automatically when using `<xsl:sort>`.
- *The XSLT stylesheet only needs to include a single line to sort content*: You'd need more code to achieve the same outcome using JavaScript arrays.

In this example, I'll include the sorting criteria in the XSLT file. The next example shows you how to apply dynamic sorting criteria using JavaScript. This example uses the resource file `planets7.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="planets7.xsl" ?>
<neighbours>
  <planet name="Venus">
    <description>
    Venus is the second planet from the sun and it has a thick layer of sulfuric
    acid clouds covering the entire planet.
    </description>
    <positionFromSun>2</positionFromSun>
    <diameter> 12104 km (7505 miles)</diameter>
    <moons> 0</moons>
    <meanTemp> 482C (900F)</meanTemp>
    <oneDay> 243.01 earth days</oneDay>
    <oneYear> 224.7 earth days</oneYear>
  </planet>
  <planet name="Mars">
    <description>
    Mars is the fourth planet from the sun and is often called the red planet.
    </description>
    <positionFromSun>4</positionFromSun>
    <diameter> 6796 km (4214 miles)</diameter>
    <moons> 2</moons>
    <meanTemp> -63C (-81F)</meanTemp>
    <oneDay> 24.62 earth hours</oneDay>
    <oneYear> 686.98 earth days</oneYear>
  </planet>
  <planet name="Mercury">
    <description>
    Mercury is the closest planet to the sun.
    </description>
    <positionFromSun>1</positionFromSun>
```

```
    <diameter> 4879 km (3025 miles)</diameter>
    <moons> 0</moons>
    <meanTemp> 179C (354F)</meanTemp>
    <oneDay> 58.65 earth days</oneDay>
    <oneYear> 87.87 earth days</oneYear>
  </planet>
</neighbours>
```

You'll notice that the first line of the XML document refers to a stylesheet called `planets7.xsl`. I've included three planets in this document and added a new element called `<positionFromSun>`.

In the previous chapter, you saw that it is possible to import and include stylesheets to avoid duplicating the same XSLT content in different files. I'll use the same approach in this chapter. The XHTML transformations appear within the file `planetsToXHTML.xsl`. You'll import this into the new stylesheet, `planets7.xsl`, which follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="planetsToXHTML.xsl"/>
  <xsl:output method="html" version="4.0" indent="yes"/>
  <xsl:template match="neighbours">
    <html>
      <head>
        <title>Sorted planets</title>
        <style type="text/css">
          body { font-family: Verdana, Arial, sans-serif; font-size:12px;}
      </style>
      </head>
      <body>
        <h1>My sorted list of planets</h1>
        <xsl:apply-templates>
          <xsl:sort select=" @name" order="descending"/>
        </xsl:apply-templates>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The stylesheet starts by importing the `planetsToXHTML.xsl` stylesheet. This stylesheet determines the display of the planets in the same way as the examples did in the previous chapter. The imported templates apply to elements that don't include a higher-priority match in the current stylesheet. As the stylesheet only matches the document element `<neighbours>`, it won't override the other declarations from the imported stylesheet. However, the `<neighbours>` template from the imported stylesheet is ignored, so you have to include the `<head>`, `<body>`, and `<child>` elements within this template.

Figure 7-1 shows `planets7.xml` displayed in IE.

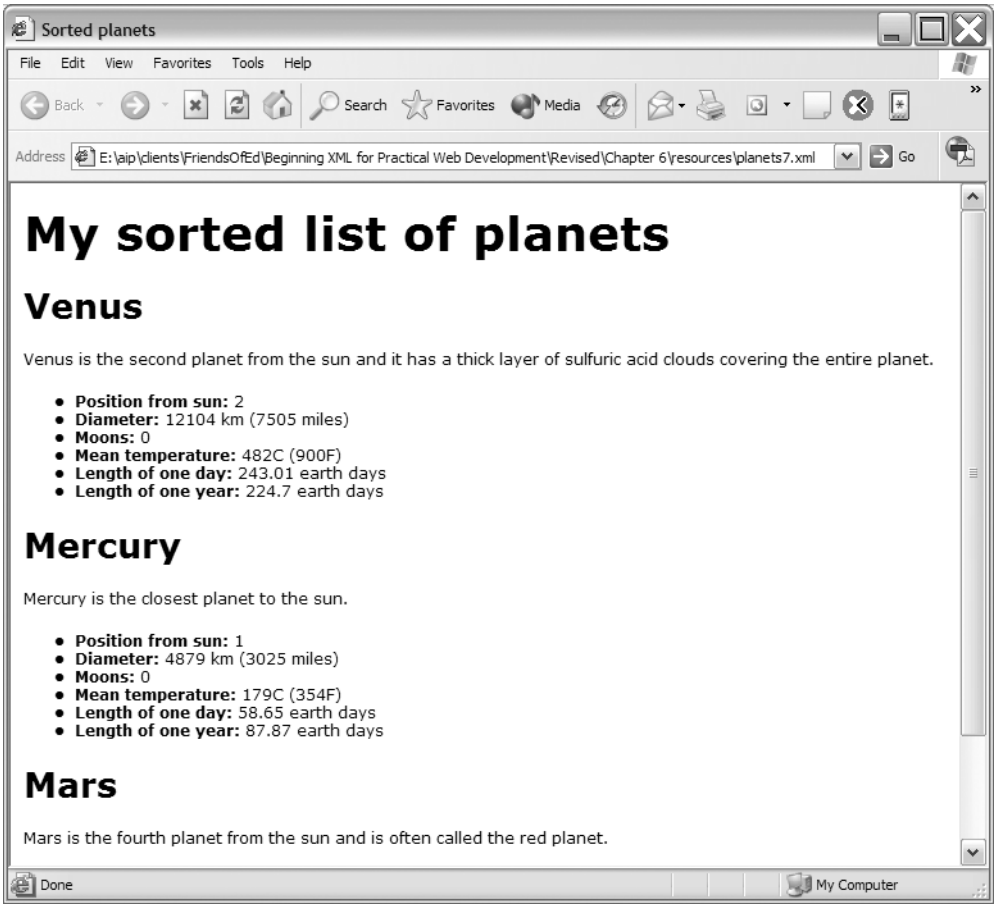

Figure 7-1. *The planets7.xml page shown in IE*

The transformation shows the planets sorted into descending alphabetical order. The stylesheet achieves this with the following line:

```
<xsl:sort select="@name" order="descending"/>
```

The `<xsl:sort>` element is applied within the `<neighbours>` element. The sorting applies to the element's children—in this case, `<planet>` elements. It finds the `name` attribute and displays each child element in descending order. You can also specify `ascending` order, which is the default if you omit the `order` attribute.

You could also sort the planets in order of their position from the sun. You can see this in the files `planets8.xml` and `planets8.xsl`. The new stylesheet includes the following sort line:

```
<xsl:sort select="positionFromSun/text()" order="ascending"/>
```

Opening the new XML file in a browser shows something similar to Figure 7-2.

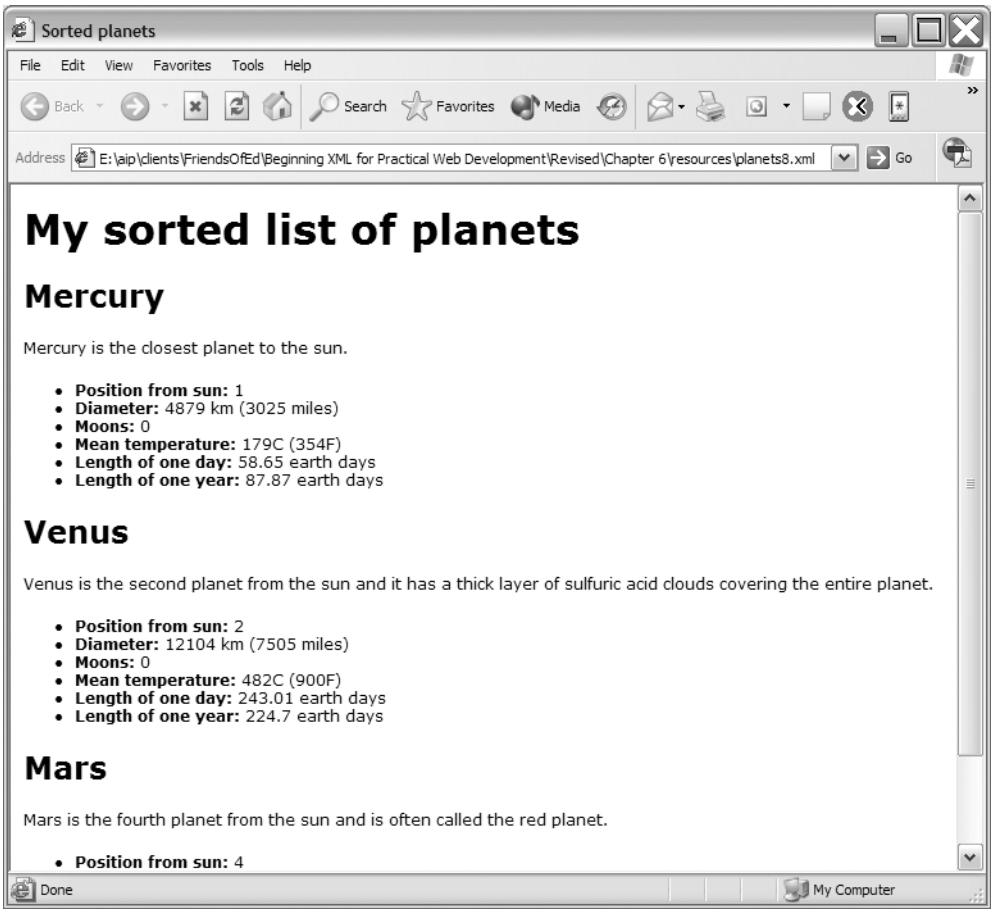

Figure 7-2. *The planets8.xml page showing a different sort order in IE*

This time, the planets display in order of their position from the sun. The sort is based on the `text()` contents within the `<positionFromSun>` child element of the `<planet>` elements. As this approach uses a text sort, you'd experience problems if you were trying to display more than 10 planets. In that case, the stylesheet could specify a numeric data type.

Stylesheets can also sort by part of the text within an element. For example, to sort in order of mean temperature from lowest to highest, a stylesheet would use the following code:

```
<xsl:sort select="substring-before(meanTemp/text(), 'C')" data-type="number"/>
```

This line uses the XPath function `substring-before` to extract all text from the `<meanTemp>` element before the `C` character. The line also specifies that the data type of the sort is `number`. Applying this sort puts the planets in order from coldest to warmest, and you can see the files `planets9.xml` and `planets9.xsl` for this example.

This example shows how easy it is to apply sorting to a complete element or part of the text within an element. One drawback is that the sort criteria are hard-coded in the XSLT stylesheet. It would be more flexible to create a dynamic sorting mechanism that allowed different types of sorts to be applied to the web page. I'll work through this example in the next section.

Sorting Dynamically with JavaScript

In this section, I'll use JavaScript to create a more dynamic sorting mechanism for the XML data. You could achieve the same outcome using server-side code to apply dynamic sorting. However, this increases the server load because you would have to reload the page with each new sort.

■ **Note** JavaScript, developed originally by Netscape, is a client-side scripting language for use with web pages. It is often used to add interactivity to a page. Because JavaScript is a client-side language, it adheres to client-side security restrictions. For example, you can't use JavaScript to create external files on the server. One use for JavaScript is to interact with XML and XHTML documents, and you'll see examples of this in the next chapter.

Dynamic sorting on the client with JavaScript and XSLT is good because:

- JavaScript can alter the sort settings and reapply the new sort transformation.
- Using JavaScript to modify the sort criteria is easier than storing the data in client-side arrays and reordering them.
- Using a client-side solution removes the overhead and processing time that would be required if using a server-side solution.

The web page needs to use slightly different JavaScript to achieve dynamic sorting for IE 6 compared with Mozilla. IE uses ActiveX controls to load and transform stylesheets, whereas Mozilla uses the TransforMiiX XSLT processor. The code in this exercise is for IE 6 only. In Chapter 8, I'll show you a JavaScript library that supports both IE and Mozilla.

This example uses the files `planets10.xml` and `planets10.xsl`. The new stylesheet follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="neighbours">
    <table border="1">
      <tr>
        <th>Name</th>
        <th>Position from sun</th>
        <th>Diameter</th>
        <th>Moons</th>
        <th>Mean temp</th>
```

```
      </tr>
      <xsl:apply-templates>
        <xsl:sort select="@name" order="ascending"/>
      </xsl:apply-templates>
    </table>
  </xsl:template>
  <xsl:template match="planet">
    <tr><td><xsl:value-of select="@name"/></td><xsl:apply-templates/></tr>
  </xsl:template>
  <xsl:template match="positionFromSun">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="diameter">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="moons">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="meanTemp">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="text()"/>
</xsl:stylesheet>
```

The stylesheet creates a page containing a table. The stylesheet matches the `<neighbours>` element in the XML document to create the `<table>` tags and headings. At the same time, it applies ascending alphabetic sorting into planet-name order:

```
<xsl:template match="neighbours">
  <table border="1">
    <tr>
      <th>Name</th>
      <th>Position from sun</th>
      <th>Diameter</th>
      <th>Moons</th>
      <th>Mean temp</th>
    </tr>
    <xsl:apply-templates>
      <xsl:sort select="@name" order="ascending"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
```

The details of each planet are displayed in a template that matches the `<planet>` element:

```
<xsl:template match="planet">
  <tr><td><xsl:value-of select="@name"/></td><xsl:apply-templates/></tr>
</xsl:template>
```

Templates match individual child elements to display their details in the table:

```
  <xsl:template match="positionFromSun">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="diameter">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="moons">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="meanTemp">
    <td><xsl:value-of select="text()"/></td>
  </xsl:template>
  <xsl:template match="text()"/>
</xsl:stylesheet>
```

Figure 7-3 shows how the table appears in IE.

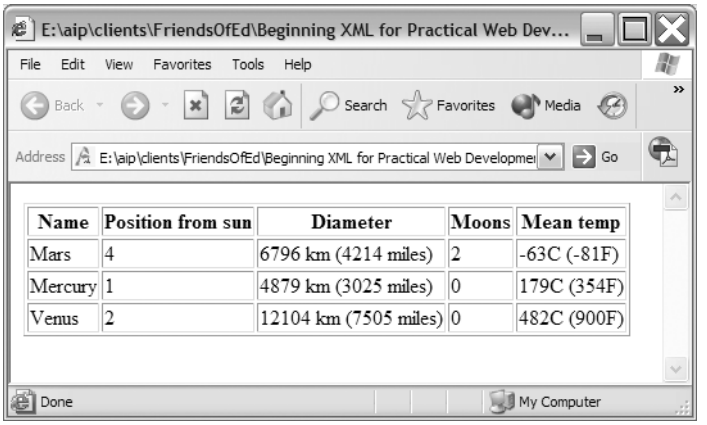

Figure 7-3. *The planets' table displayed in IE*

You might notice that the `<neighbours>` template purposely doesn't include `<html>`, `<head>`, and `<body>` elements. That's because the example will load the XML document and apply the XSLT stylesheet in a separate XHTML document. The XHTML page will display the result of the transformation in a `<div>` element. It will also add buttons that allow the user to sort the table using JavaScript. You can find this code in the new XHTML page, `sortingPlanets10.htm`:

```
<html>
  <head>
    <style>
      body {font-family: verdana, arial, sans-serif; }
      td {padding: 4px; font-size: 12px;}
    </style>
```

```
    <script language="JavaScript">
      var xmlfile = "planets10.xml";
      var xslfile = "planets10.xsl";
      var xml, xsl;
      function init() {
        xml = loadDocumentIE(xmlfile);
        xsl = loadDocumentIE(xslfile);
        doTransform();
      }
      function loadDocumentIE(filename) {
        var xmldocument = new ActiveXObject("Microsoft.XMLDOM");
        xmldocument.async = false;
        xmldocument.load(filename);
        return xmldocument;
      }
      function doTransform() {
        document.getElementById("sortoutput").innerHTML = xml.transformNode(xsl);
      }
      function orderBy(select, dataType) {
        xsl = loadDocumentIE(xslfile);
        var sortItem = xsl.getElementsByTagName("xsl:sort")[0];
        sortItem.setAttribute("select", select);
        sortItem.setAttribute("data-type", dataType);
        doTransform();
      }
  </script>
  </head>
  <body onLoad="init();">
    <h1>Table of planet information</h1>
    <div id="sortoutput">Sort output goes here</div>
    <form>
      <input type="button" onClick="orderBy('@name', 'text');"
      value="Order by name" />
      <input type="button" onClick="orderBy('positionFromSun/text()', 'number');"
      value="Order by position from the sun" />
      <input type="button" onClick="orderBy('substring-before(meanTemp/text(),➥
      \'C\');', 'number')" value="Order by mean temp" />
    </form>
  </body>
</html>
```

The code seems complicated, but I'll work through it in more detail shortly.

Open the file `sortingPlanets10.htm` in IE 6, and you should see the table of XML data, as well as three buttons. Click the buttons to sort the table. Figure 7-4 shows the page.

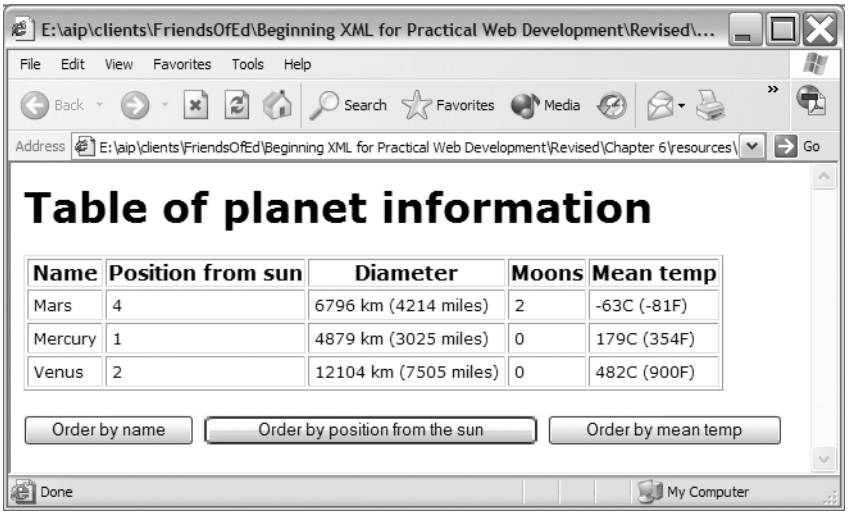

Figure 7-4. *The sortingPlanets10.htm page displayed in IE*

Let's work our way through the contents of `sortingPlanets10.htm`. The page starts with some declarations and an opening `<script>` tag:

```
<html>
  <head>
    <style>
      body {font-family: verdana, arial, sans-serif; }
      td {padding: 4px; font-size: 12px;}
    </style>
    <script language="JavaScript">
```

I'll come back to the JavaScript content.

The remainder of the page consists of layout information:

```
  <body onLoad="init();">
    <h1>Table of planet information</h1>
    <div id="sortoutput">Sort output goes here</div>
    <form>
      <input type="button" onClick="orderBy('@name', 'text');"
      value="Order by name" />
      <input type="button" onClick="orderBy('positionFromSun/text()', 'number');"
      value="Order by position from the sun" />
      <input type="button" onClick="orderBy('substring-before(meanTemp/text(),➥
      \'C\')', 'number');" value="Order by mean temp" />
    </form>
  </body>
</html>
```

The <body> section includes a header, a container for the transformed content, and a form. The form contains buttons that you can click to change the sort order. Shortly, you'll see the JavaScript that powers those buttons.

The <body> declaration includes an onLoad event handler:

```
<body onLoad="init();">
```

When the page loads, the onLoad event handler triggers the init() function, which follows:

```
function init() {
  xml = loadDocumentIE(xmlfile);
  xsl = loadDocumentIE(xslfile);
  doTransform();
}
```

The init() function calls the loadDocumentIE() function twice, loading both the XML document and the XSLT stylesheet. The function calls pass the variables xmlfile and xslfile. Those variables were defined at the beginning of the script block in the <head> section of the page:

```
var xmlfile = "planets10.xml";
var xslfile = "planets10.xsl";
```

The loadDocumentIE() loads the XML and XSLT documents:

```
function loadDocumentIE(filename) {
  var xmldocument = new ActiveXObject("Microsoft.XMLDOM");
  xmldocument.async = false;
  xmldocument.load(filename);
  return xmldocument;
}
```

The function creates an instance of the XML parser (Microsoft.XMLDOM) and references it with the variable xmldocument. The code sets the async property of the xmldocument variable to false so that the file loads synchronously—in other words, the function waits until the external XML document has finished loading before proceeding. The function finishes by using the load() method to load the specified XML document into the xmldocument variable. It returns the XML document.

The loadDocumentIE() function is called with both the XML document and XSLT stylesheet. This function can load the stylesheet because, after all, it's an XML document.

After the init() function loads both documents, it calls the doTransform() function. This function applies the XSL transformation to the XML document:

```
function doTransform() {
  document.getElementById("sortoutput").innerHTML = xml.transformNode(xsl);
}
```

The doTransform() function uses the transformNode() method of the XML parser to apply an XSLT transformation. The code passes the xsl variable to this method to specify which stylesheet to use. After the transformation, the code displays the results in the innerHTML of the sortoutput ( ) element.

■ **Note** Because the transformation is applied using JavaScript and the XML parser, planets10.xml doesn't need to include a stylesheet reference to planets10.xsl. However, I've included the reference within the XML document so you can test the transformation in a browser.

The XHTML page includes three buttons that you can click to sort the table. Clicking a button calls the orderBy() function. Each button passes the sort criteria in the function call. This includes the sorted element, as well as the type of sort to apply:

```
<input type="button" onClick="orderBy('@name', 'text');" value="Order by name" />
<input type="button" onClick="orderBy('positionFromSun/text()', 'number');"
  value="Order by position from the sun" />
<input type="button" onClick="orderBy('substring-before(meanTemp/text(), \'C\')',➥
  'number');" value="Order by mean temp" />
```

The orderBy() function follows:

```
function orderBy(select, dataType) {
  xsl = loadDocumentIE(xslfile);
  var sortItem = xsl.getElementsByTagName("xsl:sort")[0];
  sortItem.setAttribute("select", select);
  sortItem.setAttribute("data-type", dataType);
  doTransform();
}
```

The orderBy() function receives the element to sort on and its data type as parameters. The code uses these parameters to modify the XSLT stylesheet dynamically. When you click a button, the orderBy()function reloads planets10.xsl. This is required because IE makes the loaded stylesheet read-only after applying the transformation. Reloading the document allows the JavaScript to read the XSLT stylesheet again.

The code identifies the <xsl:sort> element in the stylesheet by using the getElementsByTagName() method:

```
var sortItem = xsl.getElementsByTagName("xsl:sort")[0];
```

This method returns any <xsl:sort> elements in the document. The code selects the first element by specifying index 0.

The code then sets the values of the select and datatype attributes of the <xsl:sort> element to those passed into the function. When you click the second button

```
<input type="button" onClick="orderBy('positionFromSun/text()', 'number')"
  value="Order by position from the sun" />
```

the code dynamically alters the `<xsl:sort>` element as follows:

```
<xsl:apply-templates>
  <xsl:sort select="positionFromSun/text()" data-type="number" order="ascending"/>
</xsl:apply-templates>
```

Finally, the code calls `doTransform()` to apply the altered transformation and update the `sortoutput` (`<div>`) element.

In this example, Document Object Model (DOM) scripting manipulates the elements in the stylesheet. This example touched briefly on the subject, and I'll explain it more fully in the next two chapters. In this XHTML page, JavaScript rewrote a portion of the stylesheet dynamically. You could totally rewrite the stylesheet using this method.

■ **Caution** If you have Windows XP with Service Pack 2 installed, you will run into security problems if you try to use this method to access XML files located in a different domain from the web page.

Adding Extension Functions (Internet Explorer)

If you've worked as a web developer for some time, you probably remember the days of version 3 and 4 browsers. At that time, the HTML standard wasn't consistently applied between Netscape and IE. Each browser manufacturer added nonstandard HTML tags, and there were differences in the application of existing standards. The result was that some sites had to be written in two versions—one for Netscape and one for IE.

Because of this, the XSLT specification defines a standard method of extending XSLT using extension functions and extension elements. In this example, you'll see how to create extension functions to display specific text in uppercase. I won't examine extension elements, as Microsoft XML Parser (MSXML) 3 doesn't support them. However, you could use an extension element to change the value of a variable while the stylesheet is loading.

You can use extension functions to write specific functionality. These functions are written in languages other than XSLT and are best suited to tasks such as text manipulation and disk access. They are particularly useful for quarantined environments such as intranets, where you can rely on a standard operating environment and web browser.

Although most server-side processors support extension functions, only IE supports client-side extension functions. This example only works in IE and not the Mozilla-based browsers.

In this example, I'll create JavaScript extension functions to work with the text in the `<description>` element. The example will capitalize the planet's name wherever it appears in the description. This is the same type of technique that you could use to highlight search terms within search results.

Unlike XSLT, JavaScript supports regular expressions. This allows you to specify any case for the planet's name. This example uses the resource files `planets11.xml` and `planets11.xsl`. The new stylesheet follows. I'll explain it in detail shortly:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:user="http://www.apress.com/namespace" extension-element-prefixes="msxsl">
  <xsl:import href="planetsToXHTML.xsl"/>
  <xsl:output method="html" version="4.0" indent="yes"/>
  <msxsl:script language="JScript" implements-prefix="user">
    <![CDATA[
      function capitalizeMatchingText(fullText, highlightText) {
        var reg = new RegExp(highlightText, "gi");
        var splitList = fullText.split(reg);
        return splitList.join(highlightText.toUpperCase());
      }
    ]]>
  </msxsl:script>
  <xsl:template match="planet">
    <h2>
      <xsl:value-of select="@name"/>
    </h2>
    <xsl:value-of select= ➥
    "user:capitalizeMatchingText(string(description/text()),string(@name))"/>
    <ul>
      <xsl:apply-templates/>
    </ul>
  </xsl:template>
  <xsl:template match="neighbours">
    <html>
      <head>
        <title>A simple HTML page</title>
        <style type="text/css">
          body { font-family: Verdana, Arial, sans-serif; font-size: 12px; }
      </style>
      </head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Figure 7-5 shows how this page appears in IE.

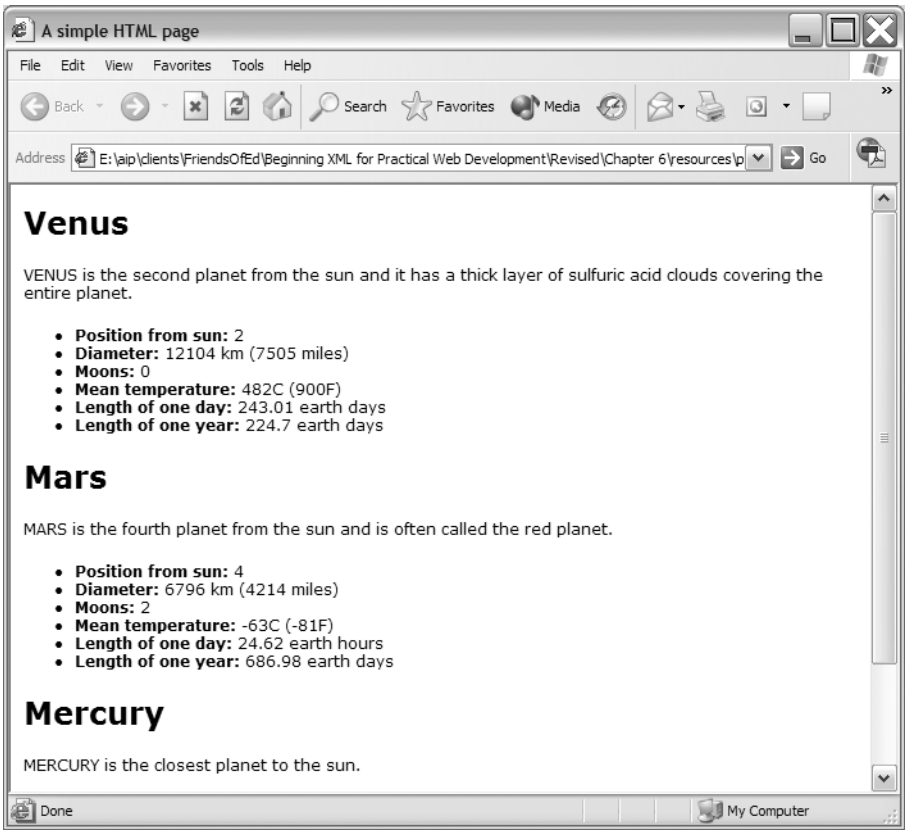

Figure 7-5. *The planets11.xml page displayed in IE*

You'll notice that instances of the planet name appear capitalized in the description.

We'll work through the code in this example in the next section. One immediate difference from the previous examples is the use of multiple namespaces. It's important to understand a little more about why these are important when working with extension functions.

Understanding More About Namespaces

Namespaces are an important concept when working with XML documents. You'll recall from earlier in the book that namespaces allow you to associate elements with a specific XML vocabulary. If you need a refresher on namespaces, you might want to reread Chapter 2.

As I mentioned, the stylesheet in this example includes two new namespace declarations. Including these namespaces allows extension functions to be added to the stylesheet:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:user="http://www.apress.com/namespace"
  extension-element-prefixes="msxsl">
```

The declarations associate the `msxsl` namespace with the `urn:schemas-microsoft-com:xslt` URI. This URI is defined by Microsoft and tells the XSLT processor to make Microsoft extension functions available to the stylesheet.

When you want to use elements from this namespace, you'll prefix them with `msxsl` in the stylesheet. The prefix `msxsl` is the convention for IE extensions; however, the text itself isn't significant. You could use any other prefix, providing that you use it consistently.

The second of the new namespace declarations defines the `user` prefix. This prefix will apply to extension functions. By convention, this namespace should be a URI referencing the organization. In this case, I've referred to an Apress URI—`http://www.apress.com/namespace`.

The URI might contain a web page describing the functions available within the namespace. However, there is no requirement for this to happen. The uniqueness of the URI is what is important here. You're not bound to use the prefix `user` and could use any other valid text.

The `<xsl:stylesheet>` element also includes the attribute

```
extension-element-prefixes="msxsl"
```

This attribute prevents the extension namespace `msxsl` from being included as output in the transformed document.

Because the declaration includes the `msxsl` namespace, the `<msxsl:script>` element is available to the stylesheet. This allows the stylesheet to include a script block containing extension functions.

```
<msxsl:script language="JScript" implements-prefix="user">
```

Notice that the `<msxsl>` element can specify the language for the script—in this case, `JScript`. The `implements-prefix="user"` attribute shows that the stylesheet will prefix the extension functions with the text `user`.

■ **Note** JScript is the Microsoft implementation of JavaScript, used with IE.

Once the stylesheet includes these namespaces, it can include extension functions within the `<msxsl:script>` element.

Adding Extension Functions to the Stylesheet

The stylesheet imports the standard stylesheet `planetsToXHTML.xsl` and sets the output method:

```
<xsl:import href="planetsToXHTML.xsl"/>
<xsl:output method="html" version="4.0" indent="yes"/>
```

The extension functions are then included in the <msxml:script> element. As I mentioned earlier, the implements-prefix attribute specifies that the word user will prefix any extension functions:

```
<msxsl:script language="JScript" implements-prefix="user">
  <![CDATA[
    function capitalizeMatchingText(fullText, highlightText) {
      var reg = new RegExp(highlightText, "gi");
      var splitList = fullText.split(reg);
      return splitList.join(highlightText.toUpperCase());
    }
  ]]>
  </msxsl:script>
```

You'll notice that a CDATA block encloses the extension function. This is necessary because the function includes the < and > characters. As an alternative, I could have used the HTML entities < or >, but using a CDATA block makes the code easier to read.

The capitalizeMatchingText()function takes two text strings—the full text to modify (fullText) and the phrase to style (highlightText). If the second string appears within the first, the function replaces the second with a capitalized version. The switch gi in the RegExp object specifies that the function will ignore the case of the highlightText string (i)and that it will do a global search (g) for all occurrences of the pattern. If you call the capitalizeMatchingText() function with the following parameters

```
capitalizeMatchingText("xml is great","Xml")
```

the function will return

```
XML is great
```

having changed the first word from lowercase to uppercase.

Although the current stylesheet imports the planetsToXHTML.xsl stylesheet, it redefines the <planet> element template to call the new JavaScript function with the following code:

```
<xsl:value-of select= ➥
  "user:capitalizeMatchingText(string(description/text()),string(@name))"/>
```

The line passes two arguments to the function: the text within the <description> element and the name attribute of the planet. The <xsl:value-of> element works with the return value from the capitalizeMatchingText() function.

Note that the code uses the XPath string() function to cast the values into text strings. If it didn't do this, it would have to convert these values into strings within the capitalizeMatchingText() function instead.

The resource files planets12.xml and planets12.xsl show the effect of calling a different function, wrapMatchingText():

```
<msxsl:script language="JScript" implements-prefix="user">
  <![CDATA[
    function wrapMatchingText(fullText, highlightText) {
      var reg = new RegExp(highlightText, "gi");
      var splitList = fullText.split(reg);
      return splitList.join("<span class='planetname'>"+highlightText+"</span>");
    }
  ]]>
</msxsl:script>
```

Instead of capitalizing the text, this function encloses it with a `<span>` tag. This tag includes a CSS class declaration. Calling the function with the parameters

```
wrapMatchingText("xml is great","Xml")
```

returns

```
<span class='planetname'>xml</span> is great".
```

Because the stylesheet generates XML output, the `<xsl:value of>` is a little different in the stylesheet:

```
<xsl:value-of disable-output-escaping="yes"
  select="user:wrapMatchingText(string(description/text()),string(@name))"/>
```

This time, the stylesheet sets the `disable-output-escaping` attribute value to `yes` because it is generating `<span>` elements. If the stylesheet left out the attribute, the angle brackets would be converted to the entities `&lt;` and `&gt;`. The `<span>` tags would then display on the page as text rather than being interpreted as XHTML elements.

The stylesheet `planets12.xsl` also includes the following CSS class declaration:

```
.planetname {background-color: #FFFF00; font-weight:bold;➥
  border: 1px solid #000000; padding: 2px;}
```

Figure 7-6 shows the transformed content using the new function. The highlight appears in a yellow color within the description, which may not be obvious from the screen shot.

GENERATING NEW XHTML TAGS

The approach shown in `planets12.xsl` is one way to generate new XHTML tags within a transformation. Although this method appears to be easy, you should use it with caution because it's easy to create documents that aren't well formed.

In Chapter 8, I'll show you how you can use the DOM to generate XML nodes rather than creating them as text. Generating XML through the DOM guarantees that the resulting content will be well formed.

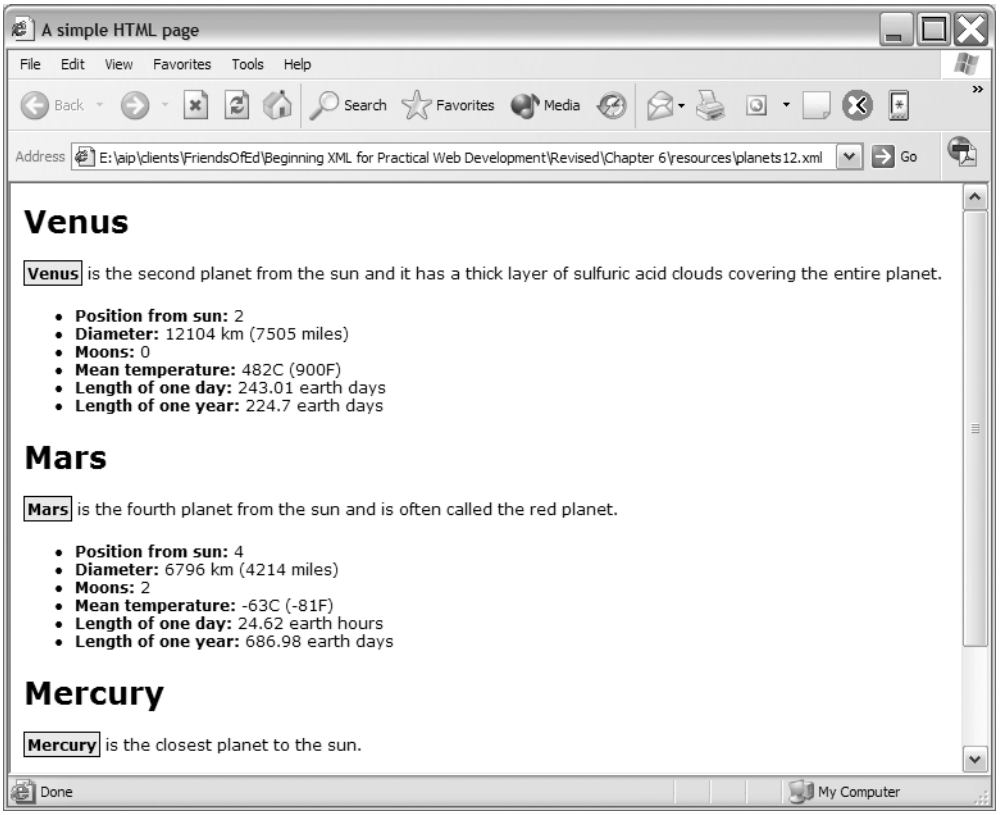

Figure 7-6. *The planets12.xml page displayed in IE*

Providing Support for Browsers Other Than IE

It would be convenient to use the same stylesheet for browsers that support extension functions and provide alternative output for other browsers. You can do this by using the `<xsl:choose>` element. This element allows you to select from one of a range of alternatives. This example checks to see if the extension function exists and calls a different transformation if necessary.

You can find this example within the files `planets13.xml` and `planets13.xsl`. The `<planet>` template from the stylesheet follows, with new lines shown in bold:

```
<xsl:template match="planet">
  <h2>
    <xsl:value-of select="@name"/>
  </h2>
  <xsl:choose>
    <xsl:when test="function-available('user:wrapMatchingText')">
      <xsl:value-of disable-output-escaping="yes"➥
      select="user:wrapMatchingText(string(description/text()),string(@name))"/>
    </xsl:when>
```

```
    <xsl:otherwise>
      <xsl:value-of select="description/text()"/>
    </xsl:otherwise>
  </xsl:choose>
  <ul>
    <xsl:apply-templates/>
  </ul>
</xsl:template>
```

The `<xsl:choose>` block provides *if, then, else* functionality to the stylesheet. It checks if the `wrapMatchingText()` function is available using a `function-available` test. If the function exists, the stylesheet calls it as before. However, if the function is unavailable, as in a non-IE browser, the stylesheet outputs the text from within the `<description>` element with no processing. Figure 7-7 shows how the page appears within both IE 6 and Firefox 1.5.

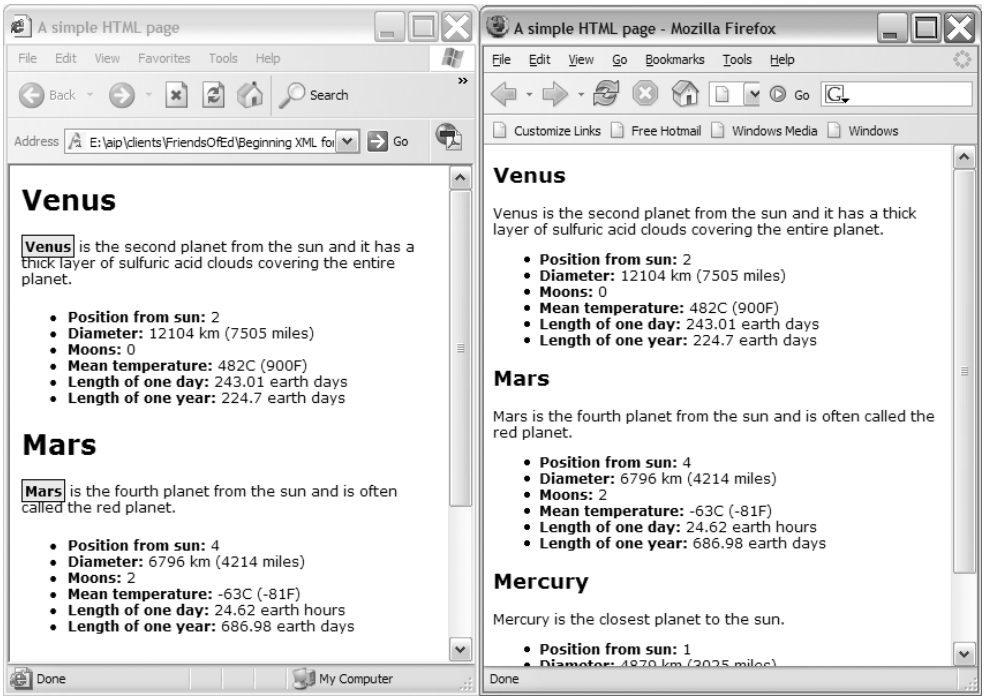

Figure 7-7. *The planets13.xml page displayed in both IE 6 and Firefox 1.5*

Working with Named Templates

Typically, in an XML-driven web site, you create a master XSLT file for the whole site and import it into other XSLT stylesheets. This manages consistency within the site, and allows for flexibility within individual sections.

The previous example imported the <planet> element template from the master stylesheet planetsToXHTML.xsl. The stylesheet duplicated the contents from the master stylesheet within planets12.xsl and planets13.xsl and edited them to introduce changes. This causes a problem if you then need to change the master stylesheet. You'd have to update the copied section each time. Using this approach would make it difficult to maintain and keep stylesheets consistent.

An alternative is to introduce a named template into the master stylesheet. You can see this approach in planets14.xml, planetsToXHTMLNamed.xsl, and planets14.xsl. The master stylesheet planetsToXHTMLNamed.xsl includes a named template. It follows with the changed lines shown in bold:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" version="4.0" indent="yes"/>
  <xsl:template match="text()"/>
  <xsl:template match="neighbours">
    <html>
      <head>
        <title>A simple HTML page</title>
        <style type="text/css">
          <xsl:call-template name="css" />
      </style>
      </head>
      <body>
        <p>
          <a href="http://www.nasa.gov/">Visit NASA!</a> |
          <a href="http://www.nineplanets.org/">Tour the solar system</a>
        </p>
        <h1>Our neighbours</h1>
        <xsl:apply-templates/>
        <hr/>
        Copyright Planetary Fun 2006.
    </body>
    </html>
  </xsl:template>
  <xsl:template name="css">
    body {font-family: Verdana, Arial, sans-serif; font-size: 12px;}
  </xsl:template>
  <xsl:template match="planet">
    <h2><xsl:value-of select="@name"/></h2>
    <xsl:value-of select="description/text()"/>
    <ul><xsl:apply-templates/></ul>
  </xsl:template>
  <xsl:template match="positionFromSun">
    <li><strong>Position from sun: </strong><xsl:value-of select="text()"/></li>
  </xsl:template>
  <xsl:template match="diameter">
    <li><strong>Diameter: </strong><xsl:value-of select="text()"/></li>
```

```
  </xsl:template>
  <xsl:template match="moons">
    <li><strong>Moons: </strong><xsl:value-of select="text()"/></li>
  </xsl:template>
  <xsl:template match="meanTemp">
    <li><strong>Mean temperature: </strong><xsl:value-of select="text()"/></li>
  </xsl:template>
  <xsl:template match="oneDay">
    <li><strong>Length of one day: </strong><xsl:value-of select="text()"/></li>
  </xsl:template>
  <xsl:template match="oneYear">
    <li><strong>Length of one year: </strong><xsl:value-of select="text()"/></li>
  </xsl:template>
</xsl:stylesheet>
```

Instead of making style declarations within the `<style>` element, the stylesheet makes a call to a named template:

```
<xsl:call-template name="css" />
```

The stylesheet also includes the template `css`:

```
<xsl:template name="css">
  body { font-family: Verdana, Arial, sans-serif; font-size: 12px; }
</xsl:template>
```

When the stylesheet processor reaches the `<xsl:call-template>` tag, it searches through all available templates to find one with a matching name. It then acts upon this template. If it can't find one, the processor will throw an error. The processor will first look through all templates in the current stylesheet and then through parent stylesheets. Bear in mind, though, that you can't import named templates.

Named templates are ideal for reducing duplicated code in stylesheets. You can easily override a named template in the current stylesheet with a further declaration using the same template name:

```
<xsl:template name="css">
  body {font-family: Verdana, Arial, sans-serif; font-size: 12px;}
  .planetname {background-color: #FFFF00; font-weight:bold; ➥
  border: 1px solid #000000; padding: 2px;}
</xsl:template>
```

If you view the `planets13.xml` document in a web browser, you won't be able to see the effect of changing the code structure. The page will render as it did previously.

The `xsl:call-template` element is a very powerful XSLT tool. You can pass parameters into a template and treat it very much like a JavaScript function with arguments. You can also use it in recursive functions. I won't cover these aspects in this book, so you may wish to explore these features further yourself.

Generating JavaScript with XSLT

In the examples so far, you've used XSLT to generate XHTML for display in a web browser. You can also use XSLT to generate output such as JavaScript. This might be useful to create web pages that are more dynamic. It also provides an alternative to using extension functions.

You can find the examples from this section in `planets14.xml` and `planets14.xsl`. Be aware that you can only apply this stylesheet in IE. The new stylesheet follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="planetName">Please select a planet</xsl:param>
  <xsl:output method="html" version="4.0" indent="no"/>
  <xsl:template match="neighbours">
    <html>
      <head>
        <title>A simple HTML page with JavaScript</title>
        <style>
      body {font-family: Verdana, Arial, sans-serif; font-size: 12px;}
    </style>
    <script language="JavaScript">
      var planetList = new Array();
      <xsl:apply-templates mode="js"/>
      function displayPlanet(name) {
        if (name!="<xsl:value-of select="$planetName"/>") {
          var w = window.open("","planetpopup", "resizable,width=400,height=300");
          w.document.open();
          w.document.write(planetList[name]);
          w.document.close();
        }
      }
    </script>
      </head>
      <body>
        <form>
        Select your planet:
        <select onChange="displayPlanet(this.options[selectedIndex].text)">
            <option>
              <xsl:value-of select="$planetName"/>
            </option>
            <xsl:apply-templates/>
          </select>
        </form>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="planet" mode="js">
    planetList["<xsl:value-of select="@name"/>"]= ➥
      '<xsl:apply-templates select="." mode="onelinehtml"/>';
```

```
  </xsl:template>
  <xsl:template match="planet" mode="onelinehtml">
    <img src="{@name}.jpg" width="100" height="100"/>
    <h2><xsl:value-of select="@name"/></h2>
    <p>
      <xsl:value-of select="normalize-space(description/text())"/>
      <br/>
      <xsl:text><hr/>Copyright Planetary Fun 2006.</xsl:text>
    </p>
  </xsl:template>
  <xsl:template match="planet">
    <option><xsl:value-of select="@name"/></option>
  </xsl:template>
</xsl:stylesheet>
```

Figure 7-8 shows what happens when you view the `planets14.xml` page in IE 6 and choose a planet from the drop-down list.

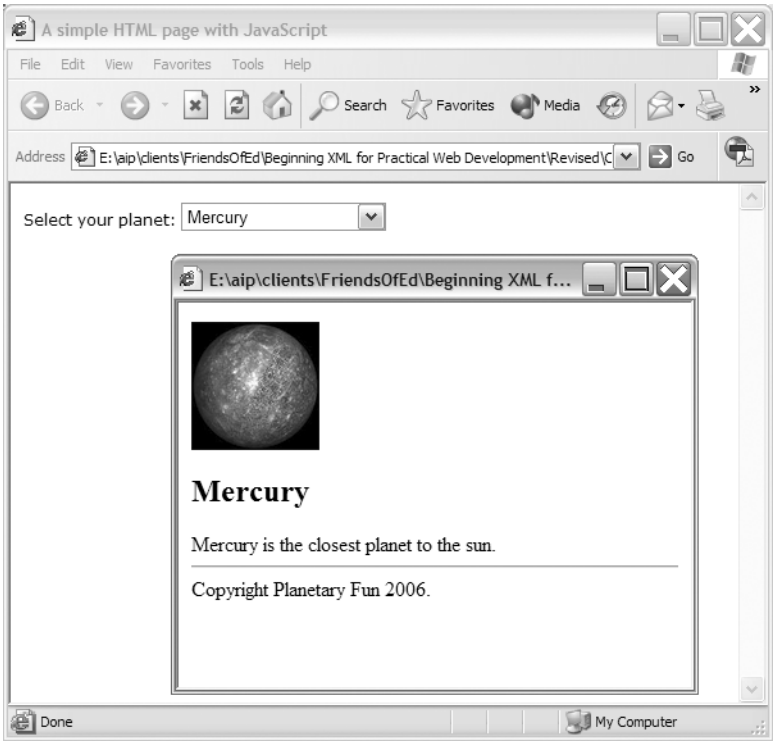

Figure 7-8. *The planets14.xml page displayed in IE*

You'll notice that the transformed content creates a list of planets in a drop-down list. When the user selects a planet from the list, a pop-up window appears showing an image and the planet's description.

Let's work through the new stylesheet to see how to achieve this effect.

Understanding XSLT Parameters

The stylesheet starts as normal with an XML declaration and an `<xsl:stylesheet>` element. The first change to the stylesheet is the introduction of a new element, `<xsl:param>`:

```
<xsl:param name="planetName">Please select a planet</xsl:param>
```

The new element is an XSLT parameter. This parameter allows the stylesheet to generate repeating content during the transformation. In this case, it defines a parameter called `planetName` that will be used as a placeholder in the drop-down list of planets. The parameter starts with the text `Please select a planet`. The stylesheet will add the other planets to the list using XSLT. The user will then be able to select any planet contained within the XML document.

You can access the value in the parameter using an `<xsl:value-of>` element and referring to the parameter name `$planetName`:

```
<xsl:value-of select="$planetName"/>
```

You'll see this a little later in the stylesheet.

As the parameter is defined at the top level of the stylesheet, it is a global parameter. Stylesheet processors can address global parameters from outside of the stylesheet. You can use JavaScript to set the parameter values.

Understanding White Space and Modes

The next line of the stylesheet sets the output for the stylesheet:

```
<xsl:output method="html" version="4.0" indent="no"/>
```

The stylesheet sets output to `html` version `4.0` for Mozilla compatibility. In previous examples, you saw the `indent` attribute set to `yes`; however, in this case, the `<xsl:output>` element sets it to `no`.

The `indent="no"` attribute allows the stylesheet to remove white space. If you don't include the declaration, the output will be indented by default to improve readability. Web browsers normally ignore white space, so it makes no difference when you output XHTML. However, white space can cause serious problems when working with JavaScript. A common problem is new line characters appearing in the middle of strings.

The stylesheet includes a template for the `<neighbours>` element. In addition to the `<head>` section and style declarations, the template creates the following JavaScript section:

```
<script language="JavaScript">
  var planetList = new Array();
  <xsl:apply-templates mode="js"/>
  function displayPlanet(name) {
    if (name!="<xsl:value-of select="$planetName"/>") {
      var w = window.open("","planetpopup", "resizable,width=400,height=300");
      w.document.open();
      w.document.write(planetList[name]);
      w.document.close();
    }
  }
</script>
```

I'll work through this JavaScript code block in detail a little later.

However, you should note that the code block starts by creating a JavaScript array called `planetList`:

```
<script language="JavaScript">
  var planetList = new Array();
```

This array will store XHTML strings relating to the planets from the XML document.

The next line

```
<xsl:apply-templates mode="js" />
```

applies templates to all elements within the current `<neighbours>` tag, where they have the matching `mode` attribute of `js`. If you look through the stylesheet, you'll see different `<planet>` templates that use the `mode` attribute. This attribute allows the stylesheet to apply different templates to the same content.

The stylesheet contains only one template for the `<planet>` elements with the mode value of `js`:

```
<xsl:template match="planet" mode="js">
  planetList["<xsl:value-of select="@name"/>"]= ➥
    '<xsl:apply-templates select="." mode="onelinehtml"/>';
</xsl:template>
```

This template generates JavaScript content for the `planetList` array.

The `js` mode template adds an entry to the `planetList` array for each `<planet>` element. The array key is the planet name, and the value comes from the `<planets>` template in `onelinehtml` mode. You'll see this template in the next section.

Incidentally, this example also includes a default `<planet>` template that doesn't have a `mode` attribute. The default template produces a list of options for the `<select>` element:

```
<xsl:template match="planet">
  <option><xsl:value-of select="@name"/></option>
</xsl:template>
```

This template will display the select box on the page.

■ **Note** The mode names don't come from a predetermined list. You can choose any mode name for your templates. This example uses descriptive names that indicate the purpose of each template.

You can see what's added to the JavaScript array by working through the `onelinehtml` template.

Working Through the onelinehtml Template

The `onelinehtml` template sets the value for each of the array items in `planetList`:

```
<xsl:template match="planet" mode="onelinehtml">
  <img src="{@name}.jpg" width="100" height="100"/>
  <h2><xsl:value-of select="@name"/></h2>
  <p>
    <xsl:value-of select="normalize-space(description/text())"/>
    <br/>
    <hr/>
    <xsl:text>Copyright Planetary Fun 2006.</xsl:text>
  </p>
</xsl:template>
```

This template creates an XHTML string that you'll ultimately display in the pop-up window.

The `src` attribute of the `<img>` tag comes from the name attribute. The stylesheet assumes that all images are named the same way—using the planet name and a `.jpeg` suffix. The `@name` expression is interpreted as XPath, as it appears within braces { }. This provides a quicker way to write an attribute value compared with the method shown in Chapter 6:

```
<xsl:attribute name="src">
  <xsl:value-of select="@name"/>.jpg
</xsl:attribute>
```

The `<p>` element contains an `<xsl:value-of>` element with the `normalize-space` function. This function strips leading and trailing white-space characters and converts multiple white-space characters to a single space. The effect is that new line characters are removed from the `<description>` element in the source XML document.

The template ends with an `<xsl:text>` element that contains the copyright text. This element writes literal text in the output, preserving white space that appears inside the element.

XSLT stylesheets ignore white space between two elements that don't contain text—for example, `<br/><hr/>`. White space between an element and text is significant. So the white space between the following two lines is significant:

```
<p><xsl:value-of select="normalize-space(description/text())"/><br /><hr/>
Copyright 2002 DinosaurOrg
```

The `<xsl:text>` element wraps the copyright text, so there are no spaces between tags and text:

```
<p><xsl:value-of select="normalize-space(description/text())"/><br/><hr/>
  <xsl:text>Copyright Planetary Fun 2006.</xsl:text>
</p>
```

Now, you can be sure that the output produced by the `onelinehtml` mode `<planet>` template won't contain white space and, therefore, won't generate JavaScript errors.

In the case of the planet Venus, the `onelinehtml` template generates the following output:

```
<img src="Venus.jpg" width="100" height="100">
<h2>Venus</h2>
<p>Venus is the second planet from the sun and it has a thick layer of sulfuric
acid clouds covering the entire planet.<br><hr>Copyright Planetary Fun 2006.</p>
```

This content appears within the `planetList` JavaScript array, as shown:

```
planetList["Venus"]= '<img src="Venus.jpg" width="100" height="100">➥
  <h2>Venus</h2><p>Venus is the second planet from the sun and it has a thick ➥
  layer of sulfuric acid clouds covering the entire planet.<br><hr>➥
  Copyright Planetary Fun 2006.</p>';
```

The code generates one array element for each planet, each containing XHTML content to display in the pop-up window.

Finishing Off the Page

The preceding section shows the effect of applying the `js` mode template with this line:

```
<xsl:apply-templates mode="js"/>
```

Remember, this line appears within the `<script>` block at the top of the page.

After the JavaScript code block uses the `js` template to add the XHTML for each planet to the `planetList` array, it defines a JavaScript function, `displayPlanet()`. The function uses the parameter defined earlier and refers to it using the variable `$planetName`:

```
function displayPlanet(name) {
  if (name!="<xsl:value-of select="$planetName"/>") {
    var w = window.open("","planetpopup", "resizable,width=400,height=300");
    w.document.open();
    w.document.write(planetList[name]);
    w.document.close();
  }
}
```

When the XSLT processor applies the XSLT stylesheet, the first line of this function transforms to

```
if (name!=" Please select a planet") {
```

In other words, the function only proceeds if the user has selected a planet. The code then creates the pop-up window

```
var w = window.open("","planetpopup", "resizable,width=400,height=300");
```

and writes the XHTML details from the `planetList` array to the document:

```
    w.document.open();
    w.document.write(planetList[name]);
    w.document.close();
  }
}
```

After the JavaScript function, the `neighbours` template creates the remainder of the XHTML page:

```
<body>
  <form>
    Select your planet:
    <select onChange="displayPlanet(this.options[selectedIndex].text)">
      <option>
        <xsl:value-of select="$planetName"/>
      </option>
      <xsl:apply-templates/>
    </select>
  </form>
</body>
```

The page consists of a form that includes a select box populated with the planet names. The `<xsl:apply-templates />` element calls the default `<planet>` template, which doesn't specify a mode. The default template creates the `<option>` elements for the `<select>` form element and uses the `planetName` parameter. You saw the XHTML file created by this stylesheet in Figure 7-8.

This example shows you how you can use a variety of XSLT techniques to create powerful and dynamic transformations. However, so far, this example will only work in IE. In the next section, we'll remedy this problem.

Generating JavaScript in Mozilla

The stylesheet that you saw in the previous example won't work properly in Mozilla because of a subtle difference between the way that IE and Mozilla treat the XSLT output. IE serializes the XML/XSLT output and reparses it as XHTML. Mozilla generates the XHTML tree directly.

In XHTML, a `<script>` element can't contain other elements, such as the `<img>` and `<p>` tags that the stylesheet generates from the `onelinehtml` template. Because IE creates the XSLT output as text and reparses it as HTML, this doesn't cause a problem.

However, using this approach in Mozilla generates JavaScript errors. Including the `<img>` and `<p>` tags in a `<script>` element isn't legal in XHTML, so the tags are ignored. The `planetList[]` array entry isn't populated correctly, generating a JavaScript error. You can avoid this problem by using `CDATA` sections and changing the way that the JavaScript function populates the array.

You can find the solution in the files `planets15.xml` and `planets15.xsl`. The amended `js` template follows:

```
<xsl:template match="planet" mode="js">
  planetList["<xsl:value-of select="@name"/>"]= ➥
    '<xsl:value-of select="@name"/>|<xsl:value-of select=➥
    "normalize-space(description/text())"/>';
</xsl:template>
```

The array is populated with two values separated by a pipe (`|`) character. The code needs to do this because it can't pass XHTML elements directly into the JavaScript array. Instead, it passes two concatenated values.

The displayPlanet() function looks quite different because it uses JavaScript to compose the XHTML tags and write them to the document:

```
function displayPlanet(name) {
  if (name!="<xsl:value-of select="$planetName"/>") {
    var w = window.open("","planetpopup", "resizable,width=400,height=300");
    var docContents = '';
    var contentArray = planetList[name].split("|");
    w.document.open();
    docContents = '<![CDATA[<img src="]]>'+ contentArray[0] + ➥
      '<![CDATA[.jpg" width="100" height="100" /><h2>]]>';
    docContents += contentArray[0];
    docContents += '<![CDATA[</h2><p>]]>';
    docContents += contentArray[1];
    docContents += '<![CDATA[<hr/>Copyright Planetary Fun 2006.</xsl:text></p>]]>';
    w.document.write (docContents)
    w.document.close();
  }
}
```

The function receives a parameter, name, that contains both the name and description of the planet, separated by a pipe (|) character. The built-in JavaScript split() function converts the string into an array called contentArray(). The first element contains the name, while the second element contains the description. The code can then write each part of the array separately to the document using document.write().

The fixed text, including XHTML elements, is wrapped in CDATA blocks and concatenated with the array content to produce output. It's a little clumsy but, when you test it, you'll find that the approach works in both IE 6 and Mozilla.

You've seen several examples showing some more advanced uses of XSLT. Now it's time to look at some tips and common troubleshooting approaches.

XSLT Tips and Troubleshooting

In this section, I want to introduce some tips for working with XSLT stylesheets. I'll also cover some techniques that you can use to troubleshoot problems that arise.

Dealing with White Space

White space is one area that can cause many headaches for new XSLT developers. If you generate only XHTML output, it's not likely to cause too many problems. As you saw with the previous example, once you start generating JavaScript, you can run into some nasty issues.

Common problems include too much white space from indenting, white space in the source document, or white space in the stylesheet. In the earlier examples, you set the indent attribute in the <xsl:output> element to yes:

```
<xsl:output method="html" version="4.0" indent="yes"/>
```

This makes it easier to read through the output from the transformation. Figure 7-9 shows the same file in IE 6, with indenting turned on (on the left) and off (on the right). The example on the left is much easier for a human to read.

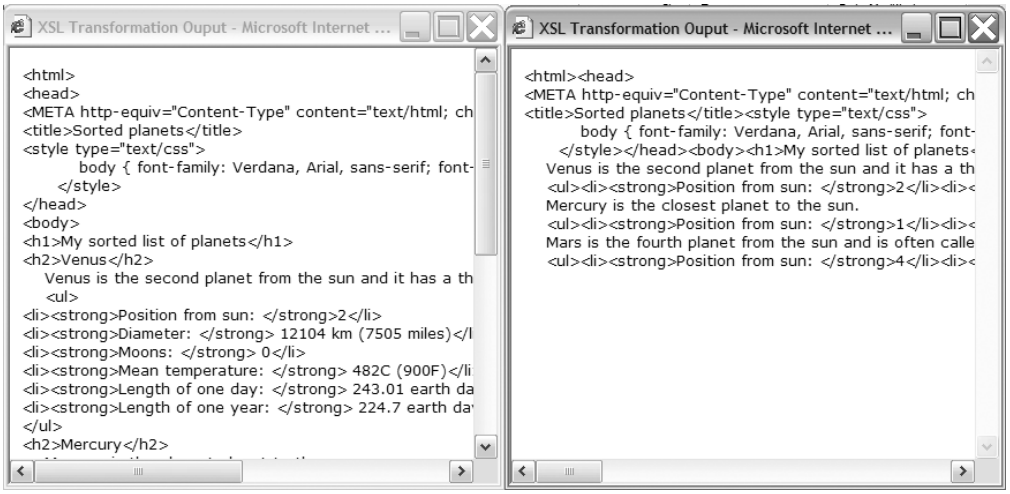

Figure 7-9. *The planets14.xml page displayed in IE*

When applying XSLT stylesheets in a web browser, indenting output can cause problems for generated JavaScript. In this case, make sure you set the value of the `indent` attribute to `no`:

```
<xsl:output indent="no"/>
```

This benefits server-side XSLT as well. Because you include less white space, the generated files are smaller.

As you saw in the previous example, you can deal with white space in the source document using the `normalize-space()` function. This function removes leading and trailing spaces, and it compresses internal white space to a single space character. You saw this within the following line:

```
<xsl:value-of select="normalize-space(description/text())"/>
```

You can also use the top-level `<xsl:strip-space>` element to strip out white-space-only nodes from elements in the source document. You can apply this to all elements with this line:

```
<xsl:strip-space elements="*" />
```

Be aware that `<xsl:strip-space>` acts on nodes that only contain white space, not nodes that include text as well as white space. The opposite is the `<xsl:preserve-space>` element, which allows you to preserve white space within a document.

As you saw in the previous example, dealing with white space in a stylesheet requires an understanding of what happens when an XSLT processor generates output. The processor removes all text nodes containing only white space, unless they're within an `<xsl:text>` element.

You can use an empty <xsl:text/> element to split text with a mixture of white space and characters into two separate text nodes:

```
<xsl:template match="planet">
  <xsl:text/>Name: <xsl:value-of select="@name"/>
</xsl:template>
```

If the stylesheet doesn't include the <xsl:text/> element, it will create white space before the text Name. Instead, the <xsl:text> element splits the white space from the text so that it is ignored. Only the text Name remains.

The <xsl:text> element also preserves white space:

```
<xsl:template match="/">
  <br/><xsl:text>
</xsl:text>
</xsl:template>
```

In this code block, using the <xsl:text> element forces a new line after the <br/> element. You could also use the entity for a new line:

```
<xsl:text>&#10;</xsl:text>
```

You can read the full details of how XSLT deals with white space at http://www.w3.org/TR/xslt#strip.

Using HTML Entities in XSLT

In XHTML, you've probably used named entities such as © and   to represent characters that don't appear on all keyboards. However, in XML, the only entities that are defined are < (<), > (>), & (&), " ("), and ' ('). You have to use the numeric form for all other entities in XML. For example, the entity & represents the ampersand (&) character.

One way to get around this is to reference entity declarations in your stylesheet:

```
<!DOCTYPE doc [
<!ENTITY e SYSTEM "entity-URI">
]>
```

Replace entity-URI with the URI for the entities that you want to include. You can then use them within your stylesheet using the normal syntax. You can find the XHTML entity definitions at http://www.w3.org/2003/entities/.

However, Mozilla does not support external entities. You can define all of the entities within the stylesheet, but that could significantly increase the size of each stylesheet. In this case, you should probably use the numeric values.

Checking Browser Type

One common role for JavaScript developers is determining the browser type of the site viewer. In XSLT 1.0 browsers, you can achieve something similar by using the system-property function to determine the vendor:

```
<xsl:value-of select="system-property('xsl:vendor')"/>
```

IE 6 returns `Microsoft`, whereas Mozilla and Netscape return `TransforMiiX`.

This should probably be a last resort when creating XSLT templates, because it's usually possible to write XSLT that works well in both IE and Mozilla.

Both IE 6 and Mozilla adhere closely to the XSLT 1.0 standard, but there are some small differences in interpretation. In general, Mozilla offers a more accurate XSLT representation than IE. This means that it's less forgiving of errors. If your stylesheet works in Mozilla, it will usually work in IE 6, but the reverse isn't always true.

If the stylesheet works when tested locally but doesn't work in Mozilla on a web server, the most likely problem is that the web server is not using a `text/xml` MIME type for serving the XML and XSLT pages. You'll need to change the web server configuration appropriately to counter this problem.

If no output appears from your stylesheet in Mozilla, even locally, then it may be that you're not generating what the browser considers valid XHTML. In order to display XHTML, the minimum output required is

```
<html><body>Some content</body></html>
```

If you include one of the `<html>` or `<body>` elements, the text of the document will appear without any XHTML markup. Without either element, nothing will appear.

The major difference between IE and Mozilla is the treatment of the XSLT output. As mentioned earlier, IE serializes the output and reparses it as XHTML. Mozilla generates the XHTML tree directly. You saw this difference in the last example, where you couldn't include XHTML elements within JavaScript arrays using XSLT. You can find more on Mozilla's XSLT support at `http://www.mozilla.org/projects/xslt/`.

Building on What Others Have Done

EXSLT (`http://www.exslt.org/`) is a community initiative to provide extensions for XSLT. The extensions are available in a number of modules on the web site, including common, math, functions, dates and times, strings, and regular expressions. Some extensions are written in pure XSLT, some use MSXML extensions so they work only in IE, and some are only for use server-side. Before creating your own functionality, you may be able to build on something from this site.

Understanding the Best Uses for XSLT

Once you start working with XSLT, you'll soon see that it is a detailed language in its own right. It can be very tempting to use it for every purpose in your XML/XHTML applications. However, XSLT works best when transforming structured data. XSLT is not good at transforming text within XML documents or styling content, and it doesn't handle calculations particularly well. You may find that the following solutions are more appropriate:

- For text formatting and styling, CSS 2 offers many useful tools and is more suited than XSLT.
- You can use extension functions for calculations if you're working in a single-browser environment such as an intranet.
- If you need to support both IE 6 and Mozilla, you may be able to use XSLT to generate client-side JavaScript that performs calculations.

Summary

In this chapter, you've worked through some of the more advanced features that you can use when working with client-side XSLT. You've learned to apply sorting with XSLT and use JavaScript to create a dynamic sorting mechanism.

You've also expanded XSLT functionality with extension functions in IE. You saw that stylesheets can check for the availability of extension functions and perform alternative transformations for non-IE browsers. You also used named templates to reduce code duplication in XSLT stylesheets. In the last example, you used XSLT to generate JavaScript. This example showed the different approaches to generating JavaScript in IE compared with Mozilla.

Finally, you've seen some of the tips and tricks for working with XSLT. The last piece of advice is that, although XSLT is powerful, it should only be used where appropriate. Other tools may be more useful.

In the next chapter, I'll discuss using browser scripting to work with XML documents. You'll see how you can use JavaScript to work with the XML DOM so that you can traverse and manipulate XML documents on the client side.