4

XSLT

So far in the book we've learned how to create well-formed XML data with a hierarchical structure. We've chosen that structure for our data, but what if we want to change that structure to suit an application or tailor our data for its users? Enter **Extensible Stylesheet Language Transformations**, **XSLT**: a language which can transform XML documents into any text-based format, XML or otherwise.

As we'll see later, XSLT is a sub-component of a larger language called XSL. XSLT is a powerful tool for e-commerce, as well as any other place where XML might be used.

As an example, suppose we're going to be running a news-oriented web site. In addition to our site itself, we'll also want to provide a way for our readers to get individual news stories from us in XML. In other words, people can use the news stories we provide in their own applications, for whatever purposes they wish; some might want to display the stories on their own web sites, others might want to put the data into their own back-end databases for some other application.

However, when we're deciding on what structure to give the XML for our news stories, we probably won't be thinking about those other uses people have for our XML; in fact, it would be impossible to guess all of the different things that people will want to do with our data. Instead, we'll be concentrating on inventing element names and a structure that properly describe a news story.

This is where XSLT comes in. Once others have received our XML documents, they can use this transformation language to transform these documents into whatever other format they wish – HTML for display on their web sites, a different XML-based structure for other applications, or even just regular text files.

XSL relies on finding parts of an XML document that match a series of predefined templates, and then applying transformation and formatting rules to each matched part. Another language, **XPath**, is used to help in finding those matching parts.

In this chapter we'll learn:

- What XSL, XSLT, and XPath are
- How XSLT, a **declarative** language, differs from **imperative** programming languages, like JavaScript
- What templates are, and how they are used

- How to address the innards of an XML document using XPath
- The main XSLT elements, and how to use them

Running the Examples

In order to perform an XSLT transformation, you need at least three things: an XML document to transform, an **XSLT stylesheet**, and an **XSLT engine**.

- The stylesheet contains the instructions for the transformation you want to perform; the "sourcecode". This is the part we'll write, and it's what this chapter is concerned with.
- The engine is the software that will carry out the instructions in your stylesheet. There are a number of engines available, many of which are free, such as:
  - Saxon, an opensource XSLT engine written in Java, available at http://saxon.sourceforge.net.
  - Xalan, which is also an opensource XSLT engine, with both C++ and Java versions offered, available at http://xml.apache.org/xalan-c/index.html and http://xml.apache.org/xalan-c/index.html, respectively.
  - MSXML, the XML parser that ships with Microsoft Internet Explorer, and is also available separately at http://msdn.microsoft.com/xml. This is the engine we'll be using for the Try It Outs in this chapter.

MSXML

MSXML, the XML parser that ships with Internet Explorer 5, includes an XSLT engine. However, IE 5 shipped before the XSLT Recommendation was finished, so the preview version of XSLT that MSXML understands is different from the Recommended XSLT. (To avoid confusion, many people call this preview version of the language "**WD-xsl**", instead of "XSLT". "WD" is short for "Working Draft", since this language is based on an earlier working draft, instead of the full XSLT Recommendation.) After the release of IE 5, Microsoft kept releasing newer versions of MSXML, which included increasing support for the XSLT recommendation, as well as some of the other W3C XML-related Recommendations.

The examples in this chapter were tested with **MSXML 3 SP1**, available at http://msdn.microsoft.com/xml. To run the examples in this chapter, you should make sure that you have this version of MSXML, or a later one.

MSXML 3 will not replace your existing MSXML parser unless you explicitly tell it to; instead, it will run "side-by-side" with the old version of MSXML, allowing you to use either one. This would be handy if you have already done development using the WD-xsl, and still want your old applications to work, however, that's probably not the case for most people. If you would like to use the new version exclusively, Microsoft provides a tool, `xmlinst.exe`, to let you run MSXML 3 in **Replace Mode**, meaning that it replaces the previous version of MSXML shipped with IE 5. It is also available from the MSDN URL above. For the examples in this book, and indeed your day-to-day work, you should install MSXML in Replace Mode, and ignore the older WD-xsl syntax. (Indeed, if you are using MSXML 4 or later, you don't have the option of using the older syntax, since support for it is removed from the parser.)

Note that IE 6 ships with MSXML 3 already installed in Replace Mode.

In addition, Microsoft also provides a tool to run MSXML's XSLT engine from the command line, called **MSXSL**. Again, it can be downloaded from the same MSDN link given above. MSXSL is run like this:

```
msxsl source stylesheet [-o output]
```

Where *source* is the XML document you want to transform, *stylesheet* is the XSLT stylesheet you are using, and *output* is the optional file you want to create, to contain the results of the transformation. If no output file is specified, the results are simply printed on the screen.

MSXSL is just a wrapper for the MSXML parser, which uses its XSLT processing capabilities. Any time that this chapter refers to MSXSL, remember that it's actually the parser/XSLT processor, MSXML, which is doing the work.

You might want to download MSXSL to a common directory, where you will be storing your XSLT stylesheets, such as `C:\XSLT`. You should also consider adding this directory to your PATH, so that you can easily run the utility from anywhere on your machine.

What is XSL?

Extensible Stylesheet Language, as the name implies, is an XML-based language used to create **stylesheets**. An XSL engine uses these stylesheets to transform XML documents into other document types, and to format the output. In these stylesheets you define the layout of the output document, and where to get the data from within the input document. That is, "retrieve data from this place in the input document; make it look like this in the output". In XSL parlance, the input document is called the **source tree**, and the output document the **result tree**.

It is important to remember that, because XSL is an XML-based language, every XSL stylesheet must be a well-formed XML document. This is a common source of confusion for novice XSLT programmers. For example, if you wanted to create a simple "hello world" stylesheet, which created an HTML document, you might do it something like this:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <p>Hello World!<br></p>
  </html>
</xsl:template>
</xsl:stylesheet>
```

We'll look at what this XSLT syntax means throughout the chapter, but it simply outputs an HTML document, that says "Hello World!". However, even though the HTML syntax used is perfectly acceptable in a browser, this is not a valid XSLT stylesheet, since it isn't well-formed XML – the `<br>` tag is not closed. (To fix this problem, you would probably change the `<br>` tag to an XML empty element, such as `<br/>`.)

There are actually two complete languages under the XSL umbrella:

- a transformation language, which is named **XSL Transformations**, or **XSLT**
- a language used to format XML documents for display, **XSL Formatting Objects**, or **XSL-FO**

Of course, the two languages can be used together, so that XSLT transforms the data, and XSL Formatting Objects then further modifies the data for display, much like Cascading Style Sheets, which will be covered in Chapter 11, *Displaying XML*.

> *XSLT became a W3C Recommendation in November of 1999; the specification can be found at* http://www.w3.org/TR/xslt. *XSL Formatting Objects was still being worked on at the time of writing, but the latest version of the specification can be found at* http://www.w3.org/TR/xsl/.

This chapter will focus on XSLT. Most XSL processors only implement the XSLT half of XSL anyway; XSLT is designed to be self-standing, meaning that developers can write XSLT engines, even if they don't want to implement the XSL Formatting Objects functionality.

Note that the output from XSLT doesn't have to be well-formed XML – it can also output HTML, or even plain text. In fact, one of the most useful aspects of XSLT is for transforming XML documents into HTML documents, for display in a browser.

Why is XSLT So Important for e-Commerce?

To get an idea of the power of XSLT, let's create a fictional example to demonstrate. Imagine that there are two companies working together, sharing information over the Internet. *Mom And Pop* is a store, which sends purchase orders to *Frank's Distribution*, which fulfills those orders. Let's assume that we have already decided that XML is the best way to send that information.

Unfortunately, the chances are that *Mom and Pop* will need a different set of information for this transaction from *Frank's Distribution*. Perhaps *Mom and Pop* needs information on the salesperson that made the order for commissioning purposes, but *Frank's Distribution* doesn't need it. On the other hand, *Frank's Distribution* needs to know the part numbers, which *Mom and Pop* doesn't really care about.

Mom and Pop uses XML such as the following:

```
<?xml version="1.0"?>
<order>
  <salesperson>John Doe</salesperson>
  <item>Production-Class Widget</item>
  <quantity>16</quantity>
  <date>
    <month>1</month>
    <day>13</day>
    <year>2000</year>
  </date>
  <customer>Sally Finkelstein</customer>
</order>
```

Whereas *Frank's Distribution* requires XML that looks more like this:

```
<?xml version="1.0"?>
<order>
  <date>2000/1/13</date>
  <customer>Company A</customer>
  <item>
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

In this scenario, we have three choices:

- *Mom and Pop* can use the same structure for its XML that *Frank's Distribution* uses. The disadvantage is that it now needs a separate XML document to accompany the first one, for its own additional information. However, the business-to-business (B2B) communication is much easier, since both companies are using the same format.
- *Frank's Distribution* can use the same format for its XML that *Mom and Pop* uses. This would have the same results as the previous choice.
- Both companies can use whatever XML format they wish internally, but transform their data to a common format whenever they need to transmit the information outside. Obviously this option provides the most flexibility for both companies, and still allows cross-company communication.

For the last option, both companies would probably agree on this common format in advance. Then, whenever *Mom and Pop* wanted to send information to *Frank's Distribution*, it would first transform it XML to this common format, and then send it. Similarly, whenever *Frank's Distribution* wanted to send information to *Mom and Pop*, it would first transform its internal XML documents to this common format, and then send it. In this way, any time either company receives information from the other, it will be in a common format.

With XSLT, this kind of transformation is quite easy: it's probably one of the most important uses of XSLT, and one of the more exciting areas where XML is already making its presence felt.

Try It Out – A Simple Business-to-Business Example

I claimed the transformation would be easy, so I guess I'd better put my money where my mouth is, and create a stylesheet that can do it!

1. Open Notepad, and type in the XML for *Mom and Pop's* data:

```
<?xml version="1.0"?>
<order>
  <salesperson>John Doe</salesperson>
  <item>Production-Class Widget</item>
  <quantity>16</quantity>
  <date>
```

```
      <month>1</month>
      <day>13</day>
      <year>2000</year>
    </date>
    <customer>Sally Finkelstein</customer>
  </order>
```

Save this as `MomAndPop.xml`.

2. Now open a new file and type in the following XSLT, which contains the instructions for the transformation:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <order>
    <date>
      <xsl:value-of select="order/date/year"/>/<xsl:value-of
      select="order/date/month"/>/<xsl:value-of select="order/date/day"/>
    </date>
    <customer>Company A</customer>
    <item>
      <xsl:apply-templates select="order/item"/>
      <quantity><xsl:value-of select="order/quantity"/></quantity>
    </item>
  </order>
</xsl:template>

<xsl:template match="item">
  <part-number>
    <xsl:choose>
      <xsl:when test=". = 'Production-Class Widget'">E16-25A</xsl:when>
      <xsl:when test=". = 'Economy-Class Widget'">E16-25B</xsl:when>
      <!--other part-numbers would go here-->
      <xsl:otherwise>00</xsl:otherwise>
    </xsl:choose>
  </part-number>
  <description><xsl:value-of select="."/></description>
</xsl:template>
</xsl:stylesheet>
```

Save this as `order.xsl`.

Looks like a big jumble of nonsense, right? Don't worry: by the end of this chapter this stylesheet will look pretty simple. For the time being, just note that this stylesheet is written in XML.

3. In order to perform the transformation, we'll get our first use of MSXSL. Open a command prompt, and find the directory where you downloaded this utility.

4. Type in `msxsl MomAndPop.xml order.xsl`, specifying the full path to the XML and XSLT files if they are in a different directory from MSXSL. For example,

   ```
   C:\XSLT\msxsl C:\input\MomAndPop.xml C:\input\order.xsl
   ```

   Your output should look like this:

```
>msxsl MomAndPop.xml order.xsl
<?xml version="1.0" encoding="UTF-8"?>
<order>
<date>2000/1/13</date>
<customer>Company A</customer>
<item>
<part-number>E16-25A</part-number>
<description>Production-Class Widget</description>
<quantity>16</quantity>
</item>
</order>
```

   which, you may notice, is the same as the *Frank's Distribution* XML document we saw earlier, with the slight addition of an `encoding` attribute in the XML declaration. *Mom And Pop* now has an XML document type that suits its business needs, but also has the ability to send the same information to *Frank's Distribution* in a format that is understandable to it.

5. Now type in `msxsl MomAndPop.xml order.xsl -o Franks.xml`, again specifying the full path to the XML and XSLT files, if need be. This time, because we have used the `-o` switch (for "output"), MSXSL won't send any output to the screen; instead, it will create a file called `Franks.xml`, which will contain the results of the transformation. Open it up in Notepad or in IE 5 or later to verify that the results are the same.

How XSLT Stylesheets Work – Templates

As mentioned earlier, XSLT stylesheets are XML documents. XSLT processors act on any XML elements that are in the XSLT namespace (which is `http://www.w3.org/1999/XSL/Transform`), and any other elements in the document, regardless of their namespace, are added to the result-tree as-is.

By convention, most people use the prefix "`xsl`" for elements in the XSLT namespace. However, as we learned in the previous chapter, you can use whatever prefix you want in your own stylesheets.

In the interest of brevity, the XSLT namespace declaration will be omitted for many of the examples in this chapter. Any time the "`xsl`" prefix is used, the XSLT namespace will be assumed.

XSLT stylesheets are built on structures called **templates**. A template specifies what to look for in the source tree, and what to put into the result tree. For example:

```
<xsl:template match="quantity">quantity tag found!</xsl:template>
```

Templates are defined using the XSLT `<xsl:template>` element. There are two important pieces to this template: the `match` attribute, and the contents of the template.

The `match` attribute specifies a pattern in the source tree; this template will be applied for any nodes in the source tree that match that pattern. In this case, the template will be applied for any elements named "`given`".

Everything inside the template, between the start- and end-tags, is what will be output to the result tree. Any elements in the XSLT namespace are special XSLT elements, which indicate to the processor that it should do some work, while any other elements or text which appear inside the `<xsl:template>` element will end up in the output document as-is.

The contents of a template can be – and almost always are – much more complex than simply outputting text. There are numerous XSLT elements you can insert into the template to perform various actions. For example, there's an `<xsl:value-of>` element, which can take information from the source tree, and add it to the result tree. The following will work the same as our previous template, but instead of outputting "`quantity tag found!`" this template will add the contents of any elements named "`quantity`" to the result tree:

```
<xsl:template match="quantity"><xsl:value-of select="."/></xsl:template>
```

In other words, if the XSLT engine finds:

```
<first>John</first>
```

it will output `John`, and if it finds:

```
<first>Andrea</first>
```

it will output `Andrea`.

We'll take a closer look at the `<xsl:template>` and `<xsl:value-of>` elements later on.

Associating Stylesheets with XML Documents Using Processing Instructions

In many, if not most, cases XSLT transformations will be performed by passing an XML document and an XSLT stylesheet to an XSLT processor. However, an XSL stylesheet can also be associated with an XML document using the same type of stylesheet processing instruction as we used for CSS in the last chapter, like this:

```
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>
```

In this case, when a browser that understands XML and XSLT loads the XML document, it will automatically do the XSLT transformation for us, and display the results.

However, since IE 5 or greater is currently the only major web browser that understands XML and XSL, this will only work for IE 5. Your stylesheets would have to use the WD-xsl syntax for XSLT – newer stylesheets won't be understood by IE 5's XSLT engine. Or, better yet, you can run MSXML in Replace Mode, as we have done above. (However, this means that any users of your application would also have to be running MSXML 3 in Replace Mode.)

Imperative Versus Declarative Programming

Before we go on to study XSLT in depth, we should stop here and discuss the type of programming you'll be doing with XSLT.

There are two classifications of programming languages in the computing world: **imperative** and **declarative**.

Imperative Programming

Imperative programming languages are ones like JavaScript, Java, or C++, where the programmer tells the computer exactly what to do, and how to do it. For example, if we had three strings, and wanted to create some HTML using those strings for paragraphs, we might write a function like the following in JavaScript:

```
function createHTML()
{
  //our three strings
  var aParagraphs = new Array("Para 1", "Para 2", "Para 3");
  var sOutput, sTemp, i;

  //create the opening HTML and BODY tags
  sOutput = new String("<html>\n<body>\n");

  for(i = 0; i < aParagraphs.length; i++)
  {
    //add a new paragraph with this string
    sTemp = new String("<p>" + aParagraphs[i] + "</P>\n");
    sOutput = sOutput + sTemp;
  }

  //add the closing HTML and BODY tags
  sOutput = sOutput + "</body>\n</html>";
  return sOutput;
}
```

This returns the following HTML:

```
<html>
<body>
<p>Para 1</p>
<p>Para 2</p>
<p>Para 3</p>
</body>
</html>
```

which is what we wanted. But in order to create the HTML we wanted with JavaScript, we had to instruct the computer to do all of the following:

- Create the `<html>` and `<body>` start-tags.
- Loop through all of the strings. For each string, create a `<p>` start-tag, add the string, and append a `</p>` end-tag. Then add this string to the main string.
- Create the `</body>` and `</html>` end-tags.

In between these steps we also had to tell the computer to add whatever new lines we needed ("`\n`" is a new line in JavaScript), and we had to make sure we did everything in the proper order. (We need to have the `<html>` start-tag before the `<p>` elements, for example.)

We had to do all of this work because JavaScript doesn't understand the concept of "HTML"; it only knows about strings. The fact that the string we created is HTML is over JavaScript's head.

Declarative Programming

XSLT is not an imperative programming language like JavaScript, it's **declarative**, and declarative languages don't require quite so much work from the developer. With XSLT, we don't specify to the computer *how* we want anything done, just *what* we want it to do. We do this using **templates,** which specify the conditions in which the process takes place and the output that's produced. How to do the work is entirely up to the processor.

For example, assuming that our three strings were in an XML file like so (`paras.xml`):

```
<?xml version="1.0"?>
<strings>
  <s>Para 1</s>
  <s>Para 2</s>
  <s>Para 3</s>
</strings>
```

we could get the same HTML as above using an XSLT stylesheet like this (`paras.xsl`):

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes" encoding="UTF-8"/>
  <xsl:template match="/">
    <html>
    <body>
      <xsl:for-each select="strings/s">
        <p><xsl:value-of select="."/></p>
      </xsl:for-each>
    </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

If you try this out in MSXSL, you'll see that we get the same results:

```
>msxsl paras.xml paras.xsl
<html>
<body>
<p>Para 1</p>
<p>Para 2</p>
<p>Para 3</p>
</body>
</html>
```

Notice that this stylesheet looks very much like the output, with some XSLT elements mixed in to specify where content should go – these are the elements in the `http://www.w3.org/1999/XSL/Transform` namespace. The XSLT engine will:

- Find the document root of the source tree. (That is, the virtual root of the document's hierarchy – more on the document root coming up.)
- Match the document root against the single template in our stylesheet.
- Output the HTML elements in the template.
- Process the XSLT elements in the template.
  - The `<xsl:for-each>` works like a template-in-a-template, and applies to any `<s>` elements which are children of a `<strings>` root element. The contents of this template are a `<P>` element, which will be output to the result tree, and the `<xsl:value-of>` element.
  - The `<xsl:value-of>` element will output the contents of those `<s>` elements to the result tree, as the contents of the `<P>` element.

Phew. That's a lot of work the XSLT engine does for us!

If we wanted to, we could even specify the order in which we want our paragraphs sorted, by simply changing the `<xsl:for-each>` element from this:

```
<xsl:for-each select="strings/s">
  <!--other XSLT here-->
</xsl:for-each>
```

to this:

```
<xsl:for-each select="/strings/s">
  <xsl:sort select="."/>
  <!--other XSLT here-->
</xsl:for-each>
```

which tells the XSLT engine to sort the results, and tells it what data to sort by. (In this case, the results are sorted alphabetically by the contents of the `<s>` element. We'll learn more about the `<xsl:sort>` element later in the chapter.)

We don't have to program any logic in our stylesheet on how to sort the strings, we simply tell it to do it, and let the XSL processor take care of the sorting for us. (Programming this logic into our JavaScript function would have required us to know complex rules about sorting algorithms, which aren't required for sorting in XSLT.)

XSLT can make things simpler for the programmer like this because it is a very specialized language – it is designed for transforming XML documents into other formats, and that's it. XSLT already knows what XML is, and it already knows what HTML is, which allows the XSLT engine to perform this work for us.

XSLT Has No Side Effects

Whenever people start talking about the benefits of XSLT, they usually mention the fact that it has **no side effects**. In order to understand why this might be considered a benefit, let's look at what they mean by "side effect".

Suppose you're writing a program in Java, or C++, or some other imperative language, which will take an XML document and print it out from a printer. Also suppose that we have a global variable in that program, which is keeping track of the current page number. Since a global variable is available to any code in the program, we can call different functions and they can update that variable, to increment it as more pages are printed.

When we call a function and it updates our global variable, this is a side effect. If any other function accesses that variable after it has been changed, that function will see the new value. This is very important, because it means we have to call the functions in a particular order. For example, if we have one function that prints out the page number, and another function that updates the page number, calling them in the wrong order would mean that the wrong page number is printed in the output.

This means two things:

- As programmers, we have to be careful to do everything in exactly the right order. If we do all the right things, but do them in the wrong order, it's just as bad as doing the wrong things!
- When we compile our code, the compiler has to run it exactly as we tell it to. That is, even if our code would run more efficiently in a different order, it has to run the way we wrote it, or else the program won't work correctly.

XSLT doesn't have these side effects. We can't modify global variables at run-time the way we can with other programming languages. This means:

- As programmers, we aren't as concerned with doing things in the proper order, just what we want the end result to be. This can greatly reduce the number of bugs in our code, since it's one less thing that we, as programmers, need to worry about.
- An XSLT engine can run the code in your stylesheet in any order it wishes. It can even run multiple pieces of code simultaneously. (Realistically, these optimizations may not exist in current XSLT engines, or may exist only to a very limited degree. However, declarative languages are inherently easier to optimize than imperative ones, so it's only a matter of time.)

There's another consequence to this: programmers who are familiar with imperative languages will have to learn a completely new style of programming. Working with, and changing, variables has been one of the most fundamental aspects of imperative programming, and it can take some getting used to working declaratively. If you're more used to declarative programming languages, XSLT as a declarative language makes a lot of sense, and allows great power and flexibility. However, if you're used to imperative languages like JavaScript, it might take a while to train yourself to stop thinking in imperative terms. Nevertheless, you can be sure that once you get used to XSLT's declarative nature, you'll begin to appreciate the power that's available to you.

XPath

Even in the short examples we've seen so far in this chapter, it seems like we're spending a lot of time looking at pieces of the source tree; we must need a pretty flexible way of pointing to different pieces of an XML document. In fact, there's a whole separate specification from the W3C for a language called **XPath**, which does exactly that: address sections of an XML document, to allow us to get the exact pieces of information we need. XSL uses XPath extensively.

XPath version 1.0 became a W3C recommendation on 16 Nov 1999, and can be found at http://www.w3.org/TR/xpath. *XPath is not only used in XSLT, it is also used in other W3C technologies, such as* ***XPointer****.*

You use **XPath expressions** to address your documents by specifying a **location path**, which conveys to XPath, step-by-step, where you are going in the document. Of course, you can't know where you're going unless you know where you are, so XPath also needs to know the **context node**; that is, the section of the XML document from which you're starting.

Think of an XPath expression as giving directions through the XML document. ("Okay, you're here, so you need to go down here, turn right at the next street, and it's the first house on the left.")

A **node** is simply a "piece" of an XML document; it could be an element, an attribute, a piece of text, or any other part of an XML document. XPath adds the concept of a node because it's often useful to work with the parent/child relationships of an XML document without having to worry about whether the branches and leaves are elements, attributes, pieces of text, or anything else. XPath also introduces the concept of a **node-set**, which is simply a collection (or "set") of nodes.

In order to illustrate the XPath examples below we'll use a sample XML document, which is a modified version of our order XML from the earlier Try It Out (`Franks.xml`), so you might want to save the following to your hard drive as `order.xml`, and keep it handy:

```
<?xml version="1.0"?>
<order OrderNumber="312597">
  <date>2000/1/13</date>
  <customer id="A216">Company A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
```

```
        <description>Production-Class Widget</description>
        <quantity>16</quantity>
      </item>
    </order>
```

We've just added some attributes, for the sake of demonstrating how to address them with XPath.

How Do Templates Affect the Context Node?

An important point to remember in XSLT is that whatever the template uses for its `match` attribute becomes the context node for that template. This means that all XPath expressions within the template are **relative** to that node. Take the following example:

```
    <xsl:template match="/order/item">
        <xsl:value-of select="part-number"/>
    </xsl:template>
```

This template will be instantiated for any `<item>` element that is a child of an `<order>` element that is a child of the document root. That `<item>` element is now the context node for this template, meaning that the XPath expression in the `<xsl:value-of>` attribute is actually only selecting the `<part-number>` element which is a child of the `<item>` element already selected for this template.

XPath Basics

As mentioned earlier, XPath expressions give directions through an XML document, from one point in the document to another. These directions are given step-by-step, starting at the context node, where each step is separated by a "`/`" character. Consider the following:

```
    order/item
```

This expression gives the following directions:

- ❑ Start at the context node (this direction is implicit – it is not actually stated in the expression)
- ❑ Go to the first child element which is named "`order`"
- ❑ Finally, select the first child element of the `<order>` element which is named "`item`"

Each step in the XPath expression specifies an XML element, in this case an `<order>` element and an `<item>` element. Alternatively, attributes can be specified in XPath expressions, by putting a "`@`" character in front of the name of the attribute. For example,

```
    order/@OrderNumber
```

is the same as the previous expression, except that instead of selecting an `<item>` child element of the `<order>` element, it is selecting an `OrderNumber` attribute of the `<order>` element.

XPath expressions are used for XSLT elements such as `<xsl:value-of>`, which is used to pull information out of the source tree, such as the following:

```
<xsl:value-of select="order/item"/>
```

which would insert the value of the `<item>` element into the result tree.

In addition to XPath expressions, there are also **XPath patterns**. In an XPath pattern, instead of giving directions to a particular node, you are specifying a *type* of node against which to match.

Patterns are used in templates, to indicate to the processor when the template should be instantiated. Consider the following:

```
<xsl:template match="order/item">
```

This template will match any `<item>` element in the source tree which is a child of an `<order>` element.

Note that I find it a lot easier to read XPath patterns backwards (from right to left). In the `<xsl:template>` above, I would read the XPath pattern as "match against any `<item>` element which is a child of an `<order>` element", whereas in the `<xsl:value-of>` earlier I would instead read the expression similar to "go to the `<order>` child of the context node, then to the `<item>` child of that `<order>` child, and select that".

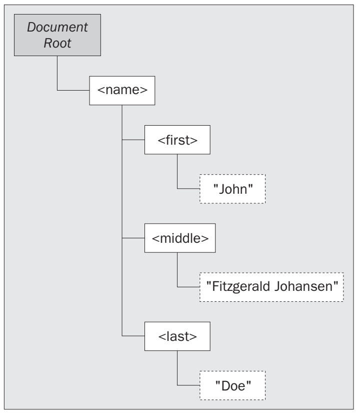
The Document Root

An important thing to note about XPath is the concept of the **document root**, which is **not** the root element we learned about in Chapter 2. Since there can be other things at the beginning or end of an XML document before or after the root element, XPath needs a virtual document root to act as the root of the document's hierarchy. In fact, the root element is a child of the document root.

Recall our `<name>` example from Chapter 2, which had this XML:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

the hierarchy according to XPath would be more like this:



The document root doesn't point at any element in the document; it's just the conceptual root of the document. In XPath, the document root is specified by a single "/". This is why the template in our first XSLT example, which matched against "/", applied to the entire document.

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

The "/" character also refers to the document root when it is at the beginning of an XPath expression or pattern. For example,

```
<xsl:value-of select="/name/first"/>
```

tells the XSLT processor to go to the document root, then to the `<name>` root element, and finally to the first child of that `<name>` element named "first", and get the value of it. This will always refer to the same element, regardless of what the context node might be, because it starts right from the document root and goes from there.

Remember: XPath expressions and patterns can be relative**, meaning that they start at the context node, or** absolute**, meaning that they start at the document root.**

The document root is especially important in XSLT stylesheets, since this is where all processing starts. The XSLT processor looks through the stylesheet for a template that matches the document root, and from this point on, all processing is controlled by the contents of this template.

If no template is supplied which matches against the document root, a default template is provided. More on default templates coming up.

Filtering XPath Expressions and Patterns

So now that we can address pieces of our XML document, can we get more specific? Of course! Not only can we locate nodes which are in specific locations in our document, but we can match against nodes with specific values, or that meet other conditions. So, for example, instead of matching against `<first>` elements from the source tree, we could match only `<first>` elements whose text is "`John`".

This is done using "`[]`" brackets, which act like a **filter** (also called a **predicate**) on the node being selected. We can test for elements with specific child nodes by simply putting the name of the child node in the "`[]`" brackets. So:

- "`order[customer]`" matches any `<order>` element which is a child of the context node and which has a `<customer>` child element.
- "`order[@number]`" matches any `<order>` element which is a child of the context node and which has a `number` attribute.
- And to take it a step further, "`order[customer = 'Company A']`" matches any `<order>` element which is a child of the context node, and which has a `<customer>` child with a value of "`Company A`".

Notice that in all of the cases previous we're selecting the `<order>` element, *not* the `number` attribute or the `<customer>` element. The "`[]`" brackets just act as a filter, so that only certain `<order>` elements will be selected.

To check for a node with a specific value, you would use "`.`" for the comparison. (In XPath, "`.`" refers to the context node.) For example, "`customer[. = 'Company A']`" matches any `<customer>` element with a value of "`Company A`".

These filters can be as complex as we want. For example, "`order[customer/@id = '216A']`" says select any `<order>` elements which have a child element named `<customer>`, which in turn has an `id` attribute with a value of "`216A`". But again, it's still the `<order>` element we're selecting, not the `<customer>` element or the `id` attribute.

Dealing with Complex Expressions

For more complex XPath expressions, like the following:

```
/order[@number = '312597']/item/part-number[@warehouse = 'Warehouse 11']
```

you might find it easier to break things down into steps. For example, you can specify the XPath expression to the node you want to select, and then add in any filters after. In this case, we can first write the XPath expression as:

```
/order/item/part-number
```

We can then narrow this down to only select the `<part-number>` elements whose `warehouse` attribute has a value of "`Warehouse 11`":

```
/order/item/part-number[@warehouse = 'Warehouse 11']
```

And finally, we select these `<part-number>` elements only if they are descended from an `<order>` element whose `number` attribute had the value "`312597`":

```
/order[@number = '312597']/item/part-number[@warehouse = 'Warehouse 11']
```

XPath Functions

In order to make XPath even more powerful and useful, there are a number of **functions** that can be used in XPath expressions. Some of these can be used to return nodes and node-sets that can't be matched with regular parent/child and element/attribute relationships. There are also functions that can be used to work with strings and numbers, which can be used both to retrieve information from the original document, and to format it for output.

You can recognize functions because they end with "`()`" parentheses. Some functions need information to work; this information is passed in the form of **parameters**, which are inserted between the "`()`" parentheses.

For example, there is a function called `string-length()`, which returns the number of characters in a string. It might be used something like this:

```
string-length('this is a string')
```

The parameter is the string "`this is a string`". The function `string-length()` will use this information to calculate its result (which, in this example, is 16, since there are 16 characters in that string).

Some functions will take parameters, but will also have **default parameters**. In these cases, you can use the function without passing it any parameters, but the function will operate as if you had passed it the default parameters. For example, the `string-length()` function defaults to the context node if no parameters are passed. So simply saying:

```
string-length()
```

is the same as saying

```
string-length(.)
```

where the period, "`.`", represents the context node.

We'll be introducing some of these XPath functions as needed throughout the chapter. For information on all of the functions available, see the XPath Recommendation, at http://www.w3.org/TR/xpath, for the core XPath functions, as well as the XSLT Recommendation, at http://www.w3.org/TR/xslt, for some extension functions added to XPath by XSLT.

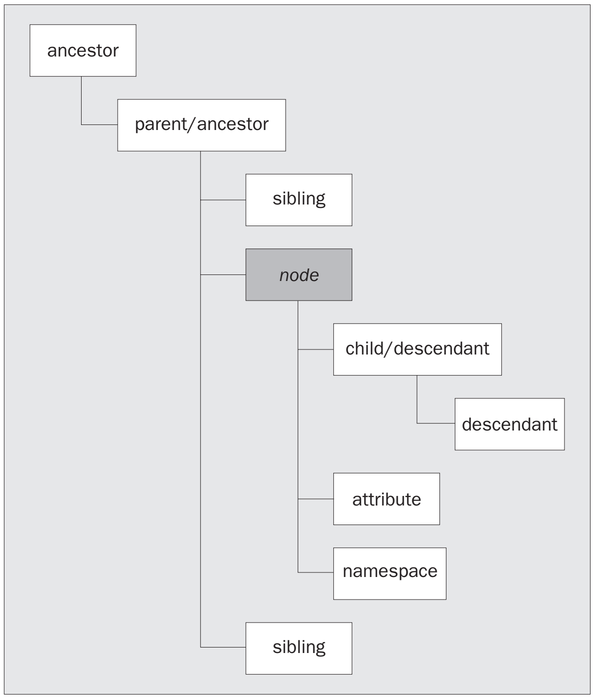
In addition, some XSLT engines allow you to define your own extension functions, to include in your stylesheets. However, the way in which you do so varies from engine to engine, meaning that any stylesheets that include such extension functions will, to some extent, be non-portable, and only work for that particular XSLT engine.

XPath Axis Names

So far we've always been dealing with children and attributes. However, there are other sections of the tree structure in XML that we could be looking at. Or, to put it another way, there are numerous directions we can travel, in addition to just going down the tree from parents to children.

The way that we move in these other directions is through different **axes**. There are 13 axes defined in XPath, which you use in XPath expressions by specifying the axis name, followed by `::`, followed by the node name. For example, "`child::order`" would specify the `<order>` element in the `child` axis, and "`attribute::OrderNumber`" would specify the `OrderNumber` attribute in the attribute `axis`.

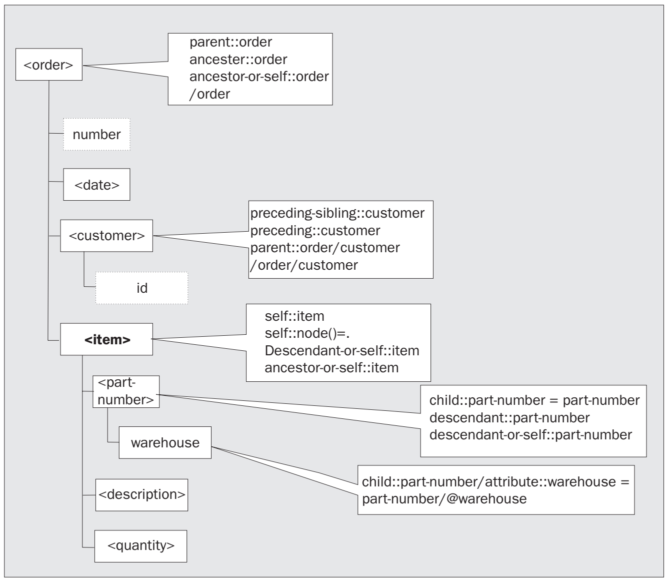
Consider the following diagram:



From our context node, we have access to parents and ancestors, children and descendants, and siblings. If the node is an element, we also have access to any attributes that are attached to that node. And the last section of the tree we have access to is the namespace. For example, consider the following XML document:

```
<?xml version="1.0"?>
<order number="312597">
  <date>2000/1/1</date>
  <customer id="216A">Customer A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

If the `<item>` element is the context node, then this diagram represents some of the ways the various nodes could be referenced, using XPath's axes:



Note that not all of the possible XPath expressions have been included; this diagram is just intended to give you a taste of what you can do with axes. Also, notice that in some cases, the axis names have shortcuts. So, for example, the "`@`" notation we've been using is a shortcut for the `attribute::` axis, and the `child::` axis name can always be omitted.

Try It Out – Using XPath Axes

To try out some of these examples, we'll create an XSLT stylesheet that selects <customer> as the context node, and then prints out values according to the XPath expression we give it.

1. Save the following XML as `xpath.xml`.(Note that this is the same as our previous `Franks.xml` document, except that a couple of attributes have been added, for the sake of addressing them in XPath):

```
<?xml version="1.0"?>
<order number="312597">
  <date>2000/1/13</date>
  <customer id="216A">Customer A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

2. Type in the following, and save it as `xpath.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="item">
  <xsl:for-each select="child::part-number">
    <xsl:value-of select="."/>
  </xsl:for-each>
</xsl:template>

<xsl:template match="text()"><!--do nothing--></xsl:template>
</xsl:stylesheet>
```

Note the `select` attribute of the `<xsl:for-each>` element. This is where you will be entering your XPath expressions. Simply replace the existing XPath expression ("`child::part-number`") with the XPath expression you wish to test.

3. Run this stylesheet through MSXSL. The results will be the value of any nodes which match the XPath expression you enter. For example, the stylesheet shown above would produce:

```
>msxsl xpath.xml xpath.xsl
E16-25A
```

This is enough to get us started using XPath in our XSLT stylesheets. As we go through the rest of this chapter, we will be learning more and more about XPath expressions and patterns, as needed.

XSLT Fundamentals

So far we've covered some of the reasons for using XSLT, and some of the general methodologies used, including XPath, which is used extensively in XSLT. You've also been teased with some sample stylesheets, with promises that all would be made clear later. Now it's time to roll up our sleeves and take a look at some of the XSLT elements we can use in our stylesheets, which will do the actual work of our XSL transformations.

This chapter won't list all of the elements available with XSLT 1.0, but will introduce the more common ones that you're likely to encounter. Furthermore, not all of the attributes are listed for some of the elements, but again, only the more common ones. For in-depth coverage of all of the XSLT elements, and their use, see the Wrox Press book, XSLT Programmer's Reference 2nd Edition, by Michael Kay, ISBN 1-861005-06-7.

<xsl:stylesheet>

The `<xsl:stylesheet>` element is the root element of nearly all XSLT stylesheets, and is used like this:

```
<xsl:stylesheet version="version number"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    exclude-result-prefixes="space-separated list of NS prefixes">
```

If you are following the current (at the time of writing) XSLT specification from the W3C, you should use "`1.0`" for the *`version number`*. Remember also that all of the XSLT elements are in the XSLT namespace, which is almost always declared on the `<xsl:stylesheet>` element, since it's the root element for the document. As mentioned before, we'll use the convention of using "`xsl`" as our namespace prefix, although you can use whatever prefix you wish. (Some people simply use "x" for their XSLT namespace prefix, to save on typing.)

Instead of the `<xsl:stylesheet>` element, you could use the `<xsl:transform>` element. It has exactly the same meaning and syntax as `<xsl:stylesheet>`, and is available for those who wish to differentiate their XSLT stylesheets from their XSL Formatting Objects stylesheets. Most people don't use it, however, and stick to `<xsl:stylesheet>` exclusively, which is the practice we'll follow.

For stylesheets which will only define one template, which matches against the document root, the `<xsl:stylesheet>` element is optional. For example, instead of this stylesheet from earlier:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
    <body>
      <xsl:for-each select="strings/s">
        <p><xsl:value-of select="."/></p>
      </xsl:for-each>
    </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

we could have used this instead:

```
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<body>
  <xsl:for-each select="strings/s">
    <p><xsl:value-of select="."/></p>
  </xsl:for-each>
</body>
</html>
```

Both of these stylesheets mean exactly the same thing to an XSL processor. All you need to specify are the `xsl:version` global attribute and the XSLT namespace prefix, and include the rest of the XSLT elements you need as usual. (Notice that the `xsl:version` attribute needs to be explicitly associated with the XSLT namespace. It is now a global attribute, as discussed in the previous chapter.) This shorthand syntax comes in handy most often when transforming XML documents to HTML, but can be useful in other transformations as well.

In fact, this shorthand was developed specifically for people who already know HTML, and who want to learn XSL. It was believed that it would be easier for them to get their feet wet before jumping right into fully-fledged XSLT.

Finally, notice that there is an `exclude-result-prefixes` attribute for this element. This is used when you are using namespace prefixes in your stylesheet, that you don't want included in the result tree. For example, consider the following XML document:

```
<name xmlns="http://www.sernaferna.com/name">John</name>
```

We might want to process this document using a stylesheet such as the following:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:in="http://www.sernaferna.com/name"
                exclude-result-prefixes="in">
<xsl:template match="/">
  <employee-name><xsl:value-of select="in:name"/></employee-name>
</xsl:template>
</xsl:stylesheet>
```

In order to get the value from the `<name>` element in our source tree, we have to specify the namespace to which it belongs – remember, we're not just dealing with a `<name>` element, but with a `<{http://www.sernaferna.com/name}name>` element. So, in our stylesheet we have declared a prefix called "`in`", which refers to that namespace, and used it in our `<xsl:value-of>`. However, by default any namespace prefixes that are declared in a stylesheet will also be declared in the result tree, which is not what we want in this case, since we don't actually use that namespace in our output. The `exclude-result-prefixes` attribute helps with this situation. It instructs the XSLT processor that, if this prefix is not used in the output, it doesn't need to be declared.

As an aside, notice that the `<name>` element in our source document used a default namespace, whereas the stylesheet used a namespace prefix to provide the same information. The XSLT stylesheet doesn't care – all it needs to know is what namespace prefix is being sought, regardless of how that namespace is declared. (By default, any namespace prefixes that are declared in your stylesheet will be included in the output, even if they aren't actually used, as in the example above.)

<xsl:template>

As we have seen already, `<xsl:template>` is the element used to define the templates which make up our XSLT stylesheets. Although this is one of the more important elements you'll be using with XSLT, it's also one of the simplest:

```
<xsl:template match="XPath pattern"
    name="template name"
    priority="number"
    mode="mode name">
```

The `match` attribute is used to specify the XPath pattern which is matched against the source tree.

The `name` attribute can be used to create a **named template**; that is, a template which you can explicitly call in your stylesheet. Named templates will be discussed in a section of their own, later in the chapter.

The `mode` attribute can be used when the same section of the source tree must be processed more than once. Modes will also be discussed in a section of their own, later in the chapter.

Template Priority

In some cases, there will be more than one template in a document which matches a particular node. In these cases, XSLT has some rules, which it uses to determine which template to call. The most important one is that a more specific template has a higher priority than a less specific one. Consider the following two templates:

```
<xsl:template match="item">
```

```
<xsl:template match="order/item">
```

An `<item>` element with any parent other than `<order>` will match the first template, but not the second, so the first template will be used. On the other hand, an `<item>` element with a parent of `<order>` will match both templates, and in this case the XSLT engine will use the second one instead, because it is more specific.

If there is more than one template which could be instantiated for a node, and XSLT's rules give them the same priority, the `priority` attribute can be used to force one to be called over the other. (The XSLT engine will instantiate the template with the highest priority.) The attribute takes a numeric value, such as "0", "0.5", or "-0.25". (Higher numbers have a higher priority, and lower numbers have a lower priority.) If two templates have exactly the same priority it is an error – however, the XSLT engine *may* still continue its processing, by choosing the template which occurs last in the stylesheet. Otherwise, it must signal an error.

<xsl:apply-templates>

The `<xsl:apply-templates>` element is used from within a template, to call other templates:

```
<xsl:apply-templates select="XPath expression"
    mode="mode name">
```

For example, consider the following:

```
<xsl:template match="order">
  <requisition>
    <xsl:apply-templates>
  </requisition>
</xsl:template>
```

This template matches `<order>` elements, so any time the XSLT engine comes across one in the source tree, this template will be instantiated. When it is, a `<requisition>` element will be inserted into the result tree. However, we also have an `<xsl:apply-templates>` element in here, meaning that the XSLT engine will then look for any children of the `<order>` element, in the source tree, which can be matched against other templates. If there are any other such templates, the content that they add to the result tree will be inserted within the `<requisition>` element's start and end tags.

If the `select` attribute is specified, then the `XPath expression` specified will be evaluated, and the result will be used as the context node which other templates are then checked against. It is optional, though, so if it's not specified then the current context node will be used instead. For example, consider the following modification of our previous template:

```
<xsl:template match="order">
  <requisition>
    <xsl:apply-templates select="item">
  </requisition>
</xsl:template>
```

In this case, the XSLT engine will only look for templates that match the `<item>` child of the `<order>` element. Even if `<order>` has other children, the XSLT engine will not try to match them against templates, so they won't get processed.

The `mode` attribute works along with the `mode` attribute of the `<xsl:template>` element, so will be discussed in the section on **Modes** later on.

Try It Out – Our First Stylesheet

We now have enough XSLT elements to create a rudimentary stylesheet. For this example, we will create an XML document containing a list of names. Our stylesheet will create a simple HTML document, and for each `<name>` encountered, it will output a paragraph with the text "`name encountered`".

This isn't really the type of processing you would probably do in the real world, but it will prepare us for some more complex transformations later in the chapter, and give us a chance to use the XSLT elements we've come across so far.

1. Here is our XML document, which you should save as `simple.xml`:

```
<?xml version="1.0"?>
<simple>
  <name>John</name>
```

```
    <name>David</name>
    <name>Andrea</name>
    <name>Ify</name>
    <name>Chaulene</name>
    <name>Cheryl</name>
    <name>Shurnette</name>
    <name>Mark</name>
    <name>Carolyn</name>
    <name>Agatha</name>
  </simple>
```

2. Now we'll create a stylesheet for this XML. First, we'll need a template to create the main HTML elements. This template will match against the document root, and then call `<xsl:apply-templates>` to let other templates take care of processing the `<name>` elements:

```
<xsl:template match="/">
  <html>
  <head><title>Sample XSLT Stylesheet</title></head>
  <body>
    <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>
```

3. Then all we need is a template to process the `<name>` elements:

```
<xsl:template match="name">
  <p>Name encountered</p>
</xsl:template>
```

4. Combining these templates together, along with an `<xsl:stylesheet>` element, produces our XSLT stylesheet in all its glory. Save this as `simple.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <head><title>Sample XSLT Stylesheet</title></head>
  <body>
    <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="name">
  <p>Name encountered</p>
</xsl:template>
</xsl:stylesheet>
```

Okay, so it's not very exciting, and neither is the output. Running this through MSXSL produces the following:

```
>msxsl simple.xml simple.xsl
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-16">
<title>sample xslt stylesheet</title></head>
<body>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
 <p>Name encountered</p>
</body>
</html>
```

Actually, on your screen the output may look more like this:

```
 < h t m l >
 < h e a d >
 < M E T A  h t t p - e q u i v  =
etc.
```

This is because the output is in the UTF-16 encoding; the Windows command prompt is treating this two-byte encoding as if it were regular ASCII text, and is outputting a space for the first byte in each two-byte character. Later we'll see how we can tell the XSLT processor to use a different encoding, such as UTF-8, which will be more readable from the command prompt.

Notice that MSXSL has added a `<META>` tag, that wasn't in the stylesheet, which gives some information about this HTML document. This is because the XSLT processor has determined that the document being produced is an HTML document. We'll see a bit more information on this later, when we discuss the `<xsl:output>` element.

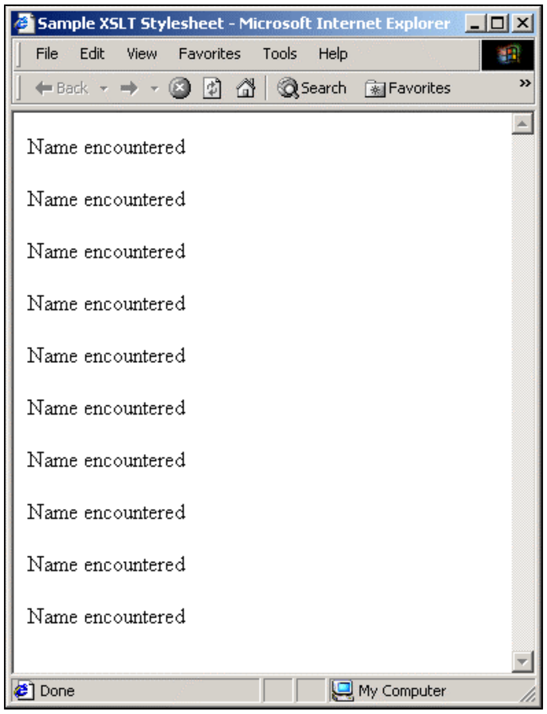
5. Remember that you can add a third parameter to MSXSL to specify the output file. So running this again like so:

```
>msxsl simple.xml simple.xsl -o simple.html
```

would produce a `simple.html` file, which would contain the result of the XSLT transformation.

MSXSL doesn't care what you name the file; it doesn't affect the XSLT processing, so the file doesn't have to have an `html` extension. We could just as easily name the file `simple.xyz`, although using the `.html` extension makes it easier to view the file in a web browser.

6. Viewing this HTML in a browser will look something like this:


Sample XSLT Stylesheet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Done
My Computer

```
Name encountered

Name encountered

Name encountered

Name encountered

Name encountered

Name encountered

Name encountered

Name encountered

Name encountered

Name encountered
```

How It Works

After the XSL processor has loaded the XML document and the XSLT stylesheet, it instantiates the template which matches the document root. This template outputs the `<html>`, `<head>`, `<body>`, and `<title>` elements, and then comes across the `<xsl:apply-templates>`.

It then searches the source tree for further nodes that can be matched against templates, and for each `<name>` element instantiates the second template. The second template simply outputs "`<p>Name encountered</p>`".

Getting Information from the Source Tree with <xsl:value-of>

Okay, so the last stylesheet was kind of boring. Now that we've got the basics for creating XSLT stylesheets, it would be great if we could actually get them to do something useful with the information in our source tree. The `<xsl:value-of>` element is the way to accomplish this:

```
<xsl:value-of select="XPath expression"
    disable-output-escaping="yes or no"/>
```

The element is simple. It searches the context node for the value specified in the `select` attribute's *`XPath expression`*, and inserts it into the result tree. For example:

```
<xsl:value-of select="customer/@id">
```

inserts the text from the `id` attribute of the `<customer>` element, and

```
<xsl:value-of select=".">
```

inserts the PCDATA from the context node into the output. (Remember, in XPath "`.`" is a shorthand for the context node.)

The `disable-output-escaping` attribute causes the XSLT processor to output "`&`" and "`<`" characters, instead of "`&amp;`" and "`&lt;`" escape sequences. Normally, the XSLT processor automatically escapes these characters for you if they make it into the output, but if you specify `disable-output-escaping` as "`yes`", then they won't. (The default is "`no`".) For example, if we have the following XML element in the input file:

```
<name>&amp;</name>
```

we have two options. This:

```
<xsl:value-of select="name" disable-output-escaping="no"/>
```

which produces:

&amp;

and this:

```
<xsl:value-of select="name" disable-output-escaping="yes"/>
```

which produces:

&

Of course, you need to keep in mind that your output will not be well-formed XML if you do this. Therefore, you should only use `disable-output-escaping` if your output is in some format other than XML (such as HTML or plain text), or if you just have no other alternative. (For example, if you need your output to contain "`&amp;`", you might put "`&amp;amp;`" into your stylesheet, with `disable-output-escaping` set to "yes". Then the XSLT engine will process the first "`&amp;`" and insert "`&`" into the stylesheet, which will then be followed by the second "`amp;`". However, this is a rare situation, so you won't often need to resort to tricks like this.)

Try It Out – <xsl:value-of> in Action

Now that we can access the information in our source tree, let's make better use of our list of names. We'll modify the stylesheet from the last Try It Out, so that in the second template each paragraph will contain the text from the `<name>` elements, instead of a simple string.

1. Open up `simple.xsl`, and change the second template to this:

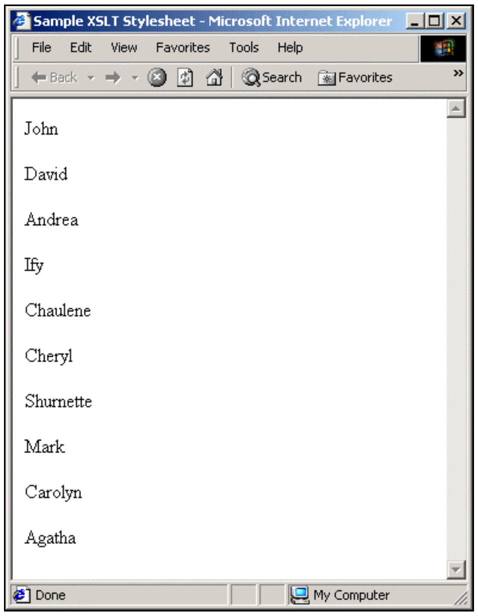
```
<xsl:template match="name">
  <P><xsl:value-of select="."/></P>
</xsl:template>
```

2. Running the modified stylesheet through MSXSL produces the following:

```
>msxsl simple.xml simple.xsl
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-16">
<title>Sample XSLT Stylesheet</title>
</head>
<body>
 <p>John</p>
 <p>David</p>
 <p>Andrea</p>
 <p>Ify</p>
 <p>Chaulene</p>
 <p>Cheryl</p>
 <p>Shurnette</p>
 <p>Mark</p>
 <p>Carolyn</p>
 <p>Agatha</p>
</body>
</html>
```

Again, there will be additional spaces interspersed in the text, as we haven't learned how to fix that yet.

3. Viewing this HTML in a browser will look something like this:


Sample XSLT Stylesheet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Done
My Computer

John

David

Andrea

Ify

Chaulene

Cheryl

Shurnette

Mark

Carolyn

Agatha

Creating the Output

So far we've seen how we can create our result tree by inserting elements and other markup into the stylesheet; anything that isn't in the XSLT namespace will be inserted as-is. However, sometimes we might need more flexibility than inserting hard-coded markup into the stylesheet. We will sometimes also need a way to tell the XSLT processor what kind of output to create – XML, HTML, or plain text.

This section will include some XSLT elements we can use to do these things.

<xsl:output>

We've mentioned that XSLT can output XML, or HTML, or even plain text. The `<xsl:output>` element allows us to specify the method we'll be using, and also gives us much more control over the way our output is created. If it is included in a stylesheet, it must be a direct child of the `<xsl:stylesheet>` element. In XSLT, elements which must be direct children of `<xsl:stylesheet>` are called **top-level** elements. The syntax of `<xsl:output>`, with some of its most commonly used attributes, is as follows:

```
<xsl:output method="xml or html or text"
   version="version"
   encoding="encoding"
   omit-xml-declaration="yes or no"
   standalone="yes or no"
   cdata-section-elements="CDATA sections"
   indent="yes or no"/>
```

The `method` attribute specifies which type of output is being produced:

- "`xml`" – the output will be well-formed XML.
- "`html`" – the output will be HTML, and will follow the rules outlined in the W3C's HTML Recommendation
- "`text`" – the output will be plain text
- Some other method. The XSLT specification says that XSLT processors are free to define other types for the `method` attribute, which they can use for other output methods. If so, the value of the `method` attribute must be a QName, so that the XSLT processor can use its namespace prefix to identify this method.

If the `<xsl:output>` element is not included, and the root element in the result tree is `<html>` (in uppercase or lowercase), then the default output method is "`html`"; otherwise, the default output method is "`xml`". When the `method` attribute is specified as "`html`", the XSL processor can do things like changing "`<br/>`" in the stylesheet to "`<br>`" in the result tree, since HTML browsers expect the `<br>` tag to be written like that. (Of course, if you wished to produce XHTML, you would probably set the `method` attribute to "`xml`", to ensure that the XSLT engine would produce well-formed XML. There isn't an "`xhtml`" value for the method attribute.) This is also why the results of our previous stylesheets, which produced HTML, included that extra `<META>` tag – the XSLT processor determined that the output type was "`html`", since the root element was `<html>`, and treated the document accordingly.

The `encoding` attribute specifies the preferred encoding to use for the result tree's text. The XSLT processor is free to ignore this attribute, if it doesn't understand the specified encoding. If the output method is "`xml`", this encoding will be included in the XML declaration. (MSXSL also includes the encoding in the `<META>` tag, if the output method is "`html`". Other XSLT processors may not include this `<META>` tag.)

The `version` and `standalone` attributes can also be used when the output method is "`xml`". Values specified for these attributes will be used to create the XML declaration for the result tree. (Note that if the `version` attribute is not included, "1.0" will be used for the XML declaration.)

For this book, we'll be specifying UTF-8 for the encoding in all of our stylesheets, because otherwise MSXSL will default the encoding to UTF-16. This will fix the problem we've been having when viewing the results of our transformations from the command line.

The `omit-xml-declaration` attribute, as its name implies, can be used when outputting XML, to specify if the XML declaration should be included. (For example, if you knew that the output of your transformation was going to be inserted as part of a larger XML document, you would want to make sure you didn't output the XML declaration, so that you wouldn't end up with two XML declarations in the final document.) The default is "`no`" when the output method is "`xml`", meaning that the XML declaration *will* appear in the result tree.

The `indent` attribute can be used to specify some formatting to be used for the result tree. If it is set to a value of "`yes`", the XSL processor is allowed to add extra whitespace to the result tree, for "pretty printing". For example, earlier we had the following XSLT stylesheet, where we specified the `<xsl:output>` element with indent set to "`yes`":

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <order>
    <date>
      <xsl:value-of select="/order/date/year"/>/<xsl:value-of
      select="/order/date/month"/>/<xsl:value-of select="/order/date/day"/>
    </date>
    <customer>Company A</customer>
    <item>
      <xsl:apply-templates select="/order/item"/>
      <quantity><xsl:value-of select="/order/quantity"/></quantity>
    </item>
  </order>
</xsl:template>

<xsl:template match="item">
  <part-number>
    <xsl:choose>
      <xsl:when test=". = 'Production-Class Widget'">E16-25A</xsl:when>
      <xsl:when test=". = 'Economy-Class Widget'">E16-25B</xsl:when>
      <!--other part-numbers would go here-->
      <xsl:otherwise>00</xsl:otherwise>
    </xsl:choose>
  </part-number>
  <description><xsl:value-of select="."/></description>
</xsl:template>
</xsl:stylesheet>
```

The result of that transformation was:

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
<date>2000/1/13</date>
<customer>Company A</customer>
<item>
<part-number>E16-25A</part-number>
<description>Production-Class Widget</description>
<quantity>16</quantity>
</item>
</order>
```

If we had left out the `<xsl:output>` element, or set `indent` to "`no`", our output would have looked like this instead:

```
>msxsl MomAndPop.xml order.xsl
<?xml version="1.0" encoding="UTF-8"?>
<order><date>2000/1/13</date><customer>Company A</customer><item><part-number>E16
-25A</part-number><description>Production-Class Widget</description><quantity>16
</quantity></item></order>
```

As far as most applications are concerned, these documents are the same. However, if a human will be reading the output, then the extra new lines are helpful. The XSLT specification gives a lot of leeway when it comes to the `indent` attribute, so different XSL processors may output the result tree differently when it is set to "`yes`", but in most cases, if not all, it will still be more easily to read by a human.

The `cdata-section-elements` attribute specifies any elements in the result tree which should be output using CDATA sections. The value of this attribute is a space-separated list of QNames of the elements that you wish to be output with CDATA sections. For example, if `cdata-section-elements` has a value of "`customer`", and the stylesheet contains an element like this:

```
<customer>A&amp;P</customer>
```

or like this:

```
<customer><![CDATA[A&P]]></customer>
```

then the output will always look like this:

```
<customer><![CDATA[A&P]]></customer>
```

Since CDATA sections are really provided as a convenience to the XML author, and to make XML documents with a lot of escaped characters more readable, you would probably only use the `cdata-section-elements` attribute for elements that you think will have a lot of escaped characters. For example, if you were authoring XHTML documents, you might do this for the `<script>` element, which is to contain scripting code.

<xsl:element>

We've already seen how to insert elements directly into the result tree, but what if we don't know ahead of time what elements we're creating? That is, what if the names of the elements that go into the result tree depend on the contents of the source tree?

The `<xsl:element>` element allows us to dynamically create elements:

```
<xsl:element name="element name"
    use-attribute-sets="attribute set names"
    namespace="namespace URI">
```

The `name` attribute specifies the name of the element. So the following:

```
<xsl:element name="blah">My text</xsl:element>
```

will produce an element like this in the result tree:

```
<blah>My text</blah>
```

Of course, this isn't very exciting, or dynamic. We could have done the same thing by simply writing that `<blah>` element into the stylesheet manually, and with less typing. The exciting and dynamic part comes into play when we change the `name` attribute like so:

```
<xsl:element name="{.}">My text</xsl:element>
```

Anything inserted into those curly braces gets evaluated as an XPath expression! In this case, we will create an element, and give it the name from the value from the context node. That is, if we have a template like this:

```
<xsl:template match="name">
  <xsl:element name="{.}">My text</xsl:element>
</xsl:template>
```

then running it against:

```
<name>Andrea</name>
```

will produce:

```
<Andrea>My text</Andrea>
```

and running it against:

```
<name>Ify</name>
```

will produce:

```
<Ify>My text</Ify>
```

The `namespace` attribute specifies what namespace, if any, this element belongs to. Like the `name` attribute, you can include curly braces, and have the namespace calculated at run-time from an XPath expression. For example, if your source tree included an element like this:

```
<some-element>http://www.sernaferna.com/namespace</some-element>
```

you could create an element like so:

```
<xsl:element name="ns-elem" namespace="{.}">...</xsl:element>
```

which would produce an element like this:

```
<ns-elem xmlns="http://www.sernaferna.com/namespace">...</ns-elem>
```

Or, if you wish to use a namespace prefix, you can use a QName for the `name` attribute, and whatever prefix you specify will be used for the namespace prefix in the output. For example,

```
<xsl:element name="x:ns-elem" namespace="{.}">...</xsl:element>
```

would produce this:

```
<x:ns-elem xmlns:x="http://www.sernaferna.com/namespace">...</x:ns-elem>
```

We'll revisit the `use-attribute-sets` attribute in the next section. First let's have a look at another example.

Try It Out – <xsl:element> in Action

To demonstrate `<xsl:element>`, let's take an XML document which uses attributes exclusively for its information, and transform it to one which uses elements instead.

1. Save the following XML as `PeopleAtts.xml`:

```
<?xml version="1.0"?>
<people>
  <name first="John" middle="Fitzgerald Johansen" last="Doe"/>
  <name first="Franklin" middle="D." last="Roosevelt"/>
  <name first="Alfred" middle="E." last="Neuman"/>
  <name first="John" middle="Q." last="Public"/>
  <name first="Jane" middle="" last="Doe"/>
</people>
```

Now let's create an XSLT stylesheet which can convert all of those attributes to elements.

2. The first step is our template to match against the document root. This template can output the root `<people>` element, and then call `<xsl:apply-templates>` to match any further elements:

```
<xsl:template match="/">
  <people>
    <xsl:apply-templates select="people/name"/>
  </people>
</xsl:template>
```

3. We then need to take care of those `<name>` elements. To do this, we can create a template to output the new `<name>` element, and then call `<xsl:apply-templates>` to take care of all of the attributes:

```
<xsl:template match="name">
  <name>
    <xsl:apply-templates select="@*"/>
  </name>
</xsl:template>
```

Notice the "`@*`" notation. The "`*`" character is an XPath **wildcard**. An XPath pattern like "`*`" means match *any element*, and a pattern like "`@*`" means match *any attribute*.

4. And finally, we need a template to process those attributes. For each attribute encountered, an element with the same name and same value should be output. This template will do the trick:

```
<xsl:template match="@*">
  <xsl:element name="{local-name()}"><xsl:value-of select="."/></xsl:element>
</xsl:template>
```

Notice that for the element name, we're using an XPath function called `local-name()`. This function will return the local part of an element's QName, without the namespace prefix. For example, for `<x:ns-elem>`, `local-name()` would return "ns-elem".

There is also a `name()` function, which will return the full QName. In this case, it would return "x:ns-elem".

5. Combining these templates together produces the following stylesheet, which you should save as `AttsToElems.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <people>
    <xsl:apply-templates select="people/name"/>
  </people>
</xsl:template>

<xsl:template match="name">
  <name>
    <xsl:apply-templates select="@*"/>
  </name>
</xsl:template>

<xsl:template match="@*">
  <xsl:element name="{local-name()}"><xsl:value-of select="."/></xsl:element>
</xsl:template>
</xsl:stylesheet>
```

6. Running it through MSXSL produces the following:

```
>msxsl PeopleAtts.xml AttsToElems.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
```

```
<last>Doe</last>
</name>
<name>
<first>Franklin</first>
<middle>D.</middle>
<last>Roosevelt</last>
</name>
<name>
<first>Alfred</first>
<middle>E.</middle>
<last>Neuman</last>
</name>
<name>
<first>John</first>
<middle>Q.</middle>
<last>Public</last>
</name>
<name>
<first>Jane</first>
<middle/>
<last>Doe</last>
</name>
</people>
```

<xsl:attribute> and <xsl:attribute-set>

Similarly to `<xsl:element>`, `<xsl:attribute>` can be used to dynamically add an attribute to an element in the result tree:

```
<xsl:attribute name="attribute name"
               namespace="namespace URI">
```

It works in exactly the same way, with the `name` attribute specifying the name of the attribute, the `namespace` attribute specifying the namespace URI (if any), and the text inside the `<xsl:attribute>` element specifying its value. For example:

```
<name><xsl:attribute name="id">213</xsl:attribute>Chaulene</name>
```

will produce the following in the result tree:

```
<name id="213">Chaulene</name>
```

and:

```
<name><xsl:attribute name="{.}">213</xsl:attribute>Chaulene</name>
```

will do the same, but the name of the attribute will be the text of the context node. Be careful, though! The name of the attribute still has to be a valid XML attribute name! So if the value of the context node above were "Fitzgerald Johansen", you would be attempting to create an attribute name with a space in it, which the XSLT Processor would reject.

Note that `<xsl:attribute>` must come before any PCDATA of the element to which it is being appended. So if we rewrote the above XSLT like this:

```
<name>Chaulene<xsl:attribute name="{.}">213</xsl:attribute></name>
```

the attribute wouldn't get appended to the `<name>` element. (MSXSL simply ignores the attribute in this case. Other processors may treat the error differently.)

Related Groups of Attributes

An `<xsl:attribute-set>` is a handy shortcut for a related group of attributes that always go together:

```
<xsl:attribute-set name="name of att set"
    use-attribute-sets="att set names">
```

For example, if we have an `id` attribute and a `size` attribute, we can define an attribute set as follows:

```
<xsl:attribute-set name="IdSize">
  <xsl:attribute name="id">A-213</xsl:attribute>
  <xsl:attribute name="size">123</xsl:attribute>
</xsl:attribute-set>
```

which will append an `id` with a value of "`A-213`" and a `size` with a value of "`123`" to any element which uses this attribute set. For example,

```
<xsl:element name="order" use-attribute-sets="IdSize"/>
```

would produce this output

```
<order id="A-213" size="123"/>
```

Or we can define the attribute set like this:

```
<xsl:attribute-set name="IdSize">
  <xsl:attribute name="{.}">A-123</xsl:attribute>
  <xsl:attribute name="size"><xsl:value-of select="."/></xsl:attribute>
</xsl:attribute-set>
```

which will append one attribute with the same name as the text of the context node, and a value of "`123`", and another value with the name `size`, and the value of the text of the context node.

This attribute set can then be appended to elements in the result tree by using the `use-attribute-sets` attribute of `<xsl:element>`. Notice also that attribute sets can use other attribute sets, since `<xsl:attribute-set>` also has a `use-attribute-sets` attribute!

Try It Out – Attributes and Attribute Sets in Action

Just for fun, let's take an XML document which uses no attributes, and transform it back to a format which uses attributes exclusively for its information.

1. Save the following document as `PeopleElems.xml`:

```
<?xml version="1.0"?>
<people>
  <name>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <name>
    <first>Franklin</first>
    <middle>D.</middle>
    <last>Roosevelt</last>
  </name>
  <name>
    <first>Alfred</first>
    <middle>E.</middle>
    <last>Neuman</last>
  </name>
  <name>
    <first>John</first>
    <middle>Q.</middle>
    <last>Public</last>
  </name>
  <name>
    <first>Jane</first>
    <middle></middle>
    <last>Doe</last>
  </name>
</people>
```

2. For our stylesheet, the first template will simply match against the document root and output the `<people>` tag, like our previous templates. But our second template, to take care of the `<name>` elements, is different:

```
  <xsl:template match="//name">
    <xsl:element name="name" use-attribute-sets="NameAttributes"/>
  </xsl:template>
```

   It creates a `<name>` element, and then uses an attribute set for the rest of the information.

   The "`//`" notation used here is a special XPath construct; it means "match any `<name>` element, anywhere in the document".

This "//" syntax is called the recursive descent operator**. There are cases, in XPath expressions, where it is useful. However, it can also greatly decrease the performance of your stylesheets. (This is because the XSLT engine must search through the entire XML document looking for matches.) As a general rule, you shouldn't use the "//" syntax unless you really need to. This template doesn't need the recursive descent operator – we could have just used "name", instead of "//name"- but I wanted to introduce it. In fact, you would never use the recursive descent operator in an XPath pattern – only in an expression.**

3. The last piece we need is the attribute set:

```
  <xsl:attribute-set name="NameAttributes">
    <xsl:attribute name="first"><xsl:value-of select="first"/></xsl:attribute>
    <xsl:attribute name="middle"><xsl:value-of select="middle"/></xsl:attribute>
    <xsl:attribute name="last"><xsl:value-of select="last"/></xsl:attribute>
  </xsl:attribute-set>
```

4. And here is the whole XSLT stylesheet that will do the transformation for us, which you should save as `ElemsToAtts.xsl`:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="/">
    <people>
      <xsl:apply-templates/>
    </people>
  </xsl:template>

  <xsl:template match="//name">
    <xsl:element name="name" use-attribute-sets="NameAttributes"/>
  </xsl:template>

  <xsl:attribute-set name="NameAttributes">
    <xsl:attribute name="first"><xsl:value-of select="first"/></xsl:attribute>
    <xsl:attribute name="middle"><xsl:value-of select="middle"/></xsl:attribute>
    <xsl:attribute name="last"><xsl:value-of select="last"/></xsl:attribute>
  </xsl:attribute-set>
</xsl:stylesheet>
```

5. Running that through MSXSL produces the following:

```
>msxsl PeopleElems.xml ElemsToAtts.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <name first="John" middle="Fitzgerald Johansen" last="Doe" />
  <name first="Franklin" middle="D." last="Roosevelt" />
  <name first="Alfred" middle="E." last="Neuman" />
  <name first="John" middle="Q." last="Public" />
  <name first="Jane" middle="" last="Doe" />
</people>
```

<xsl:text>

The `<xsl:text>` element, as its name implies, simply inserts some text (PCDATA) into the result tree.

```
<xsl:text disable-output-escaping="yes or no">
```

Much of the time the `<xsl:text>` element isn't needed, since you can just insert text directly into the stylesheet, but there are two reasons you may sometimes want to use it: it preserves all whitespace, and you can disable output escaping.

We've already seen the `disable-output-escaping` attribute, in the `<xsl:value-of>` element. We can do the same when inserting normal text into the document by using `<xsl:text>` like this:

```
<xsl:text disable-output-escaping="yes">6 is &lt; 7 &amp; 7 &gt; 6</xsl:text>
```

The text that is inserted into the result tree will look like this:

```
6 is < 7 & 7 > 6
```

(Notice that we still had to escape the characters in the XSLT stylesheet. Remember, no matter what you want the output to look like, your stylesheet is XML, and must be well-formed.)

The other advantage is that all whitespace in the `<xsl:text>` element is preserved. To see why this is important, consider the following:

```
<xsl:value-of select="'John'"/> <xsl:value-of select="'Fitzgerald Johansen'"/>
<xsl:value-of select="'Doe'"/>
```

You might expect that to produce:

```
John Fitzgerald Johansen Doe
```

However, it will actually produce:

```
JohnFitzgerald JohansenDoe
```

because the spaces between the `<xsl:value-of>` elements are stripped out by the XSLT processor. (The space in the text "Fitzgerald Johansen" is preserved, however.) If we were to insert something like this in between each one:

```
<xsl:text> </xsl:text>
```

then a space would be inserted, since the space in `<xsl:text>` is always preserved.

`<xsl:text>` can only have PCDATA for its content. It cannot contain other XSLT elements.

Default Templates

I mentioned earlier that there are some built-in XSLT templates. Let's take a look at those templates, so that you won't be surprised when the XSLT processor starts using them.

If you don't supply a template in your document which matches against the document root, XSLT provides a default one, which simply then applies any other templates which exist. This default template is defined as follows:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

This matches against any elements in the document, or the document root, and calls `<xsl:apply-templates>`, to process any children. There is also a built-in template for text and attribute nodes, which is:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

This template simply adds the value of the text node or attribute to the result tree. The net result of combining these two default templates is that you could create a stylesheet with no templates of your own, and the defaults would go through every element and attribute in the source tree, and print out the values.

Finally, there is a default template for PIs (Processing Instructions) and comments, which does nothing. It is defined as:

```
<xsl:template match="processing-instruction()|comment()"/>
```

Similarly, the XSLT Recommendation states that there is a built-in template for namespace nodes, which also does nothing. However, there is no pattern that can match against a namespace node, so you can't override this template with one of your own. If you need information about a namespace prefix or URI, you will need to get it out of the source tree manually, using `<xsl:value-of>` or a similar construct.

An important thing to remember about the default templates is that they are always lower priority than the templates you define in your stylesheets. This means that the defaults will only be used if there are no other templates defined for a particular node.

Try It Out – Making Use of the Built-In Templates

Let's create a quick example, to see how default templates work.

1. We'll use our familiar `<name>` example, which you probably already have saved from previous chapters as `name.xml`:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

2. Our first crack at this stylesheet will be simple; all we want it to do is print out the value of the `<first>` element. Save the following as `default.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="first">
  First name:  <xsl:value-of select="."/>
</xsl:template>
</xsl:stylesheet>
```

3. Run this through MSXSL. You will get the following results:

```
>msxsl name.xml default.xsl


 First name:  John
 Fitzgerald Johansen
 Doe
```

We didn't supply a template for the document root, so the default was applied, which is what we wanted. However, the default was also applied for text nodes in our document, which is *not* what we wanted. We can fix that by adding one more template to our stylesheet. Modify `default.xsl` as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="first">
  First name:  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="text() | @*">
  <!--do nothing-->
</xsl:template>
</xsl:stylesheet>
```

4. Run this through MSXSL again. This time you should get the proper results:

```
>msxsl name.xml default.xsl

 First name:  John
```

How It Works

When the XSLT processor begins processing this XML document, the first thing it does is look for a template that matches the document root. However, in this stylesheet, there is no such template, so the XSLT processor uses the default template instead. All this template does is call `<xsl:apply-templates>`, so the XSLT processor will look at each child of the document root, and look for templates to instantiate.

In this case, the only child of the document root is the `<name>` element. Once again, however, there is no template which matches a `<name>` element, so the default template is called, which simply calls `<xsl:apply-templates>` again.

This time, when `<xsl:apply-templates>` is called, there are a number of children: the `<first>`, `<middle>`, and `<last>` elements, and the whitespace text nodes in between all of these elements. In our first crack at the stylesheet, the XSLT processor was applying the default template for text nodes to the whitespace in our `<name>` element, while in the second iteration we overrode the default template, to print out nothing for text and attribute nodes. I included an XML comment in this template, to make it explicit that I didn't want to process these nodes. (Of course, our source document doesn't have any attributes, but for a generic template like this it's often better to be safe than sorry.)

Next, the XSLT processor comes to the `<first>` element, and the results are obvious: it instantiates our template, which prints out some text and the value of the element.

And finally, the XSLT processor will process our `<middle>` and `<last>` elements. Because we have defined no templates for these elements, the default will again be applied, which will call `<xsl:apply-templates>`. Each of these elements contains only a text node. In the first iteration of our stylesheet, this text node was being handled by the default, and therefore being printed into the result tree, whereas our second iteration of the stylesheet overrode the default template, and kept this text from being added to the result.

Try It Out – Default Template Gotchas

Although the default templates can be very helpful in your stylesheets, they can also cause unexpected results, when you're not careful. For example, in step 2 of the Try It Out above we created a stylesheet to print out the value from the `<first>` element, but because of the default templates, we ended up with the values from *all* of the elements being printed out. Creating templates for specific elements that you want to ignore will be a common thing to do in XSLT, so that you override the defaults.

The defaults can also cause you problems when you're debugging your stylesheets, as this Try It Out will demonstrate.

1. For this Try It Out, we'll create a stylesheet to process the `order.xml` document, and produce a text report of the information contained therein.

2. Create the following stylesheet, and save it to your hard drive as `defaulttemplates.xsl`:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8" />

<xsl:template match="order">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="date">
  Date:  <xsl:value-of select="."/>
</xsl:template>
```

```
<xsl:template match="customers">
  Customer:  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="item">
  Part number:  <xsl:value-of select="part-number"/>
  Description:  <xsl:value-of select="description"/>
  Quantity:  <xsl:value-of select="quantity"/>
</xsl:template>
</xsl:stylesheet>
```

3. Run the file through MSXSL. You will get results similar to the following:

```
>msxsl order.xml defaults.xsl


 Date:  2000/1/13
 Customer A

 Part number:  E16-25A
 Description:  Production-Class Widget
 Quantity:  16
```

Notice that, instead of "Customer: Customer A" in our output, we just get "Customer A". Why is this? Unfortunately, we have misspelled the name "`customer`" in our stylesheet, and called it "`customers`" instead. This means that this template doesn't match any elements in the source tree, and will never get called. However, since XSLT provides a default template, it is processing the `<customer>` element, and appending its contents to the result tree for us!

This is potentially very confusing, and has caused me personally untold hours of grief in my own stylesheets when I have made similar mistakes! Because of the way our output is structured, it looks like our template is getting called, even though it's not. If that template were more complicated, I would probably assume that the problem was in the template itself, and start trying to simplify it to find the problem, before I would even realize that the template wasn't getting called in the first place.

4. The solution to this problem is simple, of course: change the `match` attribute of the third template from "`customers`" to "`customer`". You will then get the proper output:

```
>msxsl order.xml defaults.xsl


 Date:  2000/1/13

 Customer:  Customer A

 Part number:  E16-25A
 Description:  Production-Class Widget
 Quantity:  16
```

Controlling the Flow of Stylesheets

As you are creating your stylesheets, you may sometimes need to decide at run-time how the results should be created. This section will introduce some XSLT elements that you can use to control the flow of processing within your stylesheets.

Conditional Processing with <xsl:if> and <xsl:choose>

Every programming language in the world has to let you make choices in your code, otherwise it wouldn't be very useful. XSLT is no exception, and it allows us a couple of ways to make choices: `<xsl:if>` and `<xsl:choose>`:

```
<xsl:if test="Boolean expression">
```

```
<xsl:choose>
  <xsl:when test="Boolean expression">
  <xsl:when test="Boolean expression">
  <xsl:otherwise>
</xsl:choose>
```

For both `<xsl:if>` and `<xsl:choose>`, the Boolean expression is simply an XPath expression which is converted to a Boolean value.

Boolean values can only have one of two choices: True or False, yes or no, on or off, 1 or 0, etc.

The expression is converted to a Boolean according to the following rules:

- If the value is numeric, it's considered `false` if it is 0. If the number is any other value, positive or negative, it is `true`.
- If the value is a string, it is `true` if its length is longer than 0 characters.
- If the value is a node-set, it is `true` if it's not empty, otherwise it's `false`.
- Any other type of object is converted to a Boolean in a way that is dependent on the type of object.

For example, this expression would be considered `true` if there were a `<name>` element that is a child of the context node, otherwise it would be `false`:

```
<xsl:if test="name">
```

`<xsl:if>` is the simpler of the two conditional processing constructs. It evaluates the expression in the `test` attribute, and if it is True the contents of the `<xsl:if>` element are evaluated. If the test expression is not True, the contents of `<xsl:if>` are not evaluated. For example, consider the following:

```
<xsl:if test="name">Name encountered.</xsl:if>
```

If there is a `<name>` element that is a child of the context node, then the text "`Name encountered.`" will be inserted into the result tree. If there is no `<name>` child of the context node, then nothing will happen.

Note that `<xsl:if>` does not change the context node, the way a template `match` does. That is, even if the `test` expression evaluated as `true`, and we entered the `<xsl:if>` element, `<name>` would not be the context node; the context node would still be whatever it was before we evaluated the `test` expression.

`<xsl:choose>` provides a bit more flexibility than `<xsl:if>`. It allows us to make any one of a number of choices, and even allows for a "default" choice, if desired. For example:

```
<xsl:choose>
  <xsl:when test="salary[number(.) &gt; 2000]">A big number</xsl:when>
  <xsl:when test="salary[number(.) &gt; 1000]">A medium number</xsl:when>
  <xsl:otherwise>A small number</xsl:otherwise>
</xsl:choose>
```

If the `<number>` element contains a numeric value which is greater than 2000, then the string "`A big number`" will be inserted into the result tree. If the number is greater than 1000, the string "`A medium number`" will be inserted, and in any other case the string "`A small number`" will be inserted. (Notice that we use an XPath function, `number()`, to convert the value in the `<number>` element to a numeric value. Remember, all of the text in an XML document is just that – text. XSLT won't treat the data as anything else until we tell it to. If we hadn't used the `number()` function, and let XSLT treat the text as text, then any value would evaluate to True, as long as it was greater than 0 characters.)

In this code fragment, we escaped the ">" signs using "`&gt;`". This isn't strictly necessary, since ">" characters are allowed in XML as-is; however, "<" characters are not, and this often trips people up when writing stylesheets. Many people consider it a good practice to always escape both ">" and "<" characters, just to be consistent.

Note that for the `<xsl:otherwise>` to be evaluated, any of the following could be true:

- `<number>` could contain a numeric value, which is less than 1000
- There could be no `<number>` node where we specified it in our test expression
- `<number>` could contain text, instead of a numeric value. If the `number()` function is passed text which isn't a number, such as "John", it returns the special "`NaN`" value, which stands for "Not a Number". `NaN` evaluates to `false`, when treated as a Boolean.

That is, none of the other tests in the stylesheet can pass the `test` attribute, for any reason.

In an `<xsl:choose>`, as soon as one of the test expression evaluates to `true`, and the XSLT processor is done executing the `<xsl:when>`, control leaves the `<xsl:choose>`. So even if two of the `<xsl:when>` test expressions would evaluate to `true`, only the first one will be processed, and control will leave the `<xsl:choose>` without evaluating the second `<xsl:when>`. If we had written our example like this instead:

```
<xsl:choose>
  <xsl:when test="number[number(.) > 1000]">A medium number</xsl:when>
  <xsl:when test="number[number(.) > 2000]">A big number</xsl:when>
  <xsl:otherwise>A small number</xsl:otherwise>
</xsl:choose>
```

then the second `<xsl:when>` would *never* get called, since any numbers which are greater than 1000 would cause the first `<xsl:when>` to be `true`.

The `<xsl:otherwise>` is not mandatory in an `<xsl:choose>`. If it is not included, and none of the `<xsl:when>` test expressions evaluates to `true`, then control will leave the `<xsl:choose>` without inserting anything into the result tree.

Try It Out – Conditional Processing in Action

Let's imagine that we have a database of a company's employee information. The database can return the information in XML form, like so:

```
<?xml version="1.0"?>
<employee FullSecurity="1">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555)555-5555<extension>2974</extension></phone>
  <salary>62,000</salary>
  <area>3</area>
</employee>
```

The `FullSecurity` attribute is determined by the ID in the database of the user who requested the information: "`1`" indicates a member of H.R., who has access to the salary information, and "`0`" indicates someone who has no such access. (A truly poor use of security, but helpful for demonstrating conditional processing.)

The company is broken down into various physical locations, so the `<area>` element tells us which physical location this employee is stationed at.

1. We don't have an actual database, so we'll just save the above XML as `employee.xml`, and pretend that it came from a database. (We'll be looking at this for real in Chapter 13.)

2. Now let's create an HTML document which can display this employee's information. Our first template will do most of the work in creating this document; it will create the HTML layout, and insert the values for all of our information except for the phone number and the area. It will also determine if it is allowed to insert the salary, depending on the security.

```
<xsl:template match="/">
  <html>
  <head>
  <title><xsl:value-of select="employee/name"/></title>
  </head>
  <body>
  <h1>Employee: <xsl:value-of select="employee/name"/></h1>
```

```
    <p>Department: <xsl:value-of select="employee/department"/></p>
    <p><xsl:apply-templates select="employee/phone"/></p>
    <xsl:if test="number(employee/@FullSecurity)">
      <p>salary: <xsl:value-of select="employee/salary"/></p>
    </xsl:if>
    <p>Location: <xsl:apply-templates select="employee/area"/></p>
    </body>
    </html>
  </xsl:template>
```

Most of this is pretty easy to us by now, so the only portion of the code we've highlighted is the section that determines whether we can output the salary. The value of the `FullSecurity` attribute is just text, so we have to first convert it to a number, using the `number()` XPath function, as we discussed earlier. Then, since the `test` attribute expects a Boolean value, it converts that number to `true` or `false`. In our case, we have decided to set `FullSecurity` to either 1 (which evaluates to `true` in XPath), or 0 (which evaluates to `false`).

3. Because the phone number is a little more complex, we'll create a simple template to process just that:

```
<xsl:template match="phone">
  Phone: <xsl:value-of select="text()"/> x<xsl:value-of select="extension"/>
</xsl:template>
```

Notice that we used an XPath function, `text()`, for the XPath expression in our `<xsl:value-of>`. This function returns the text of the context node. So why didn't we just use `"."`?

The answer is that when we say `<xsl:value-of select="."/>`, XSLT inserts the text of not only the context node, but also the text of any descendant elements. In this case, that would be "(555)555-55552974", which is not what we want. Luckily, `text()` only returns the text from the context node, and not the descendant nodes, so we can specify exactly what text we want.

4. And finally, we need a template to take care of that `<area>` element. It needs to decide where this employee is located, based on the value in the XML document. We can use `<xsl:choose>` to do that, like this:

```
<xsl:template match="area">
  <xsl:choose>
    <xsl:when test=". = '1'">Toronto</xsl:when>
    <xsl:when test=". = '2'">London</xsl:when>
    <xsl:when test=". = '3'">New York</xsl:when>
    <xsl:when test=". = '4'">Tokyo</xsl:when>
    <xsl:otherwise>Unknown Location</xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

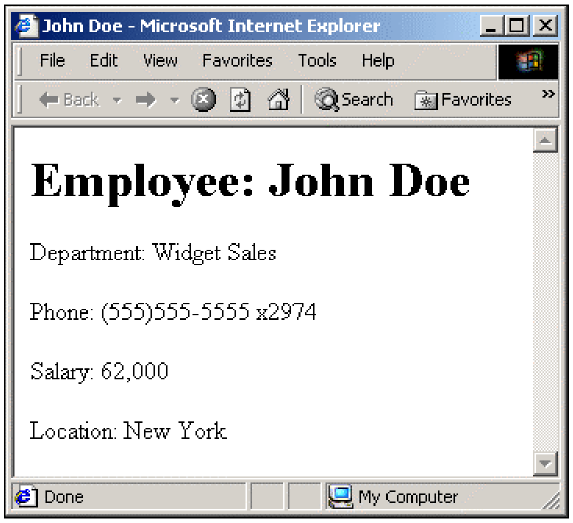
5. Save the final stylesheet as `EmployeeToHTML.xsl`, which will look like this:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" encoding="UTF-8"/>

<xsl:template match="/">
  <html>
  <head>
  <title><xsl:value-of select="employee/name"/></title>
  </head>
  <body>
  <h1>Employee: <xsl:value-of select="employee/name"/></h1>
  <p>Department: <xsl:value-of select="employee/department"/></p>
  <p><xsl:apply-templates select="employee/phone"/></p>
  <xsl:if test="number(employee/@FullSecurity)">
    <p>Salary: <xsl:value-of select="employee/salary"/></p>
  </xsl:if>
  <p>Location: <xsl:apply-templates select="employee/area"/></p>
  </body>
  </html>
</xsl:template>

<xsl:template match="phone">
  Phone: <xsl:value-of select="text()"/> x<xsl:value-of select="extension"/>
</xsl:template>

<xsl:template match="area">
  <xsl:choose>
    <xsl:when test="number(.) = 1">Toronto</xsl:when>
    <xsl:when test="number(.) = 2">London</xsl:when>
    <xsl:when test="number(.) = 3">New York</xsl:when>
    <xsl:when test="number(.) = 4">Tokyo</xsl:when>
    <xsl:otherwise>Unknown Location</xsl:otherwise>
  </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

6. When MSXSL is run against the XML above, this stylesheet produces the following HTML:

```
>msxsl employee.xml EmployeeToHTML.xsl
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>john doe</title>
</head>
<body>
<h1>Employee: John Doe</h1>
<p>Department: Widget Sales</p>
<p>
  Phone: (555)555-5555 x2974</p>
<p>Salary: 62,000</p>
<p>Location: New York</p>
</body>
</html>
```

which looks like this in a web browser:


John Doe - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Done
My Computer

Employee: John Doe

Department: Widget Sales

Phone: (555)555-5555 x2974

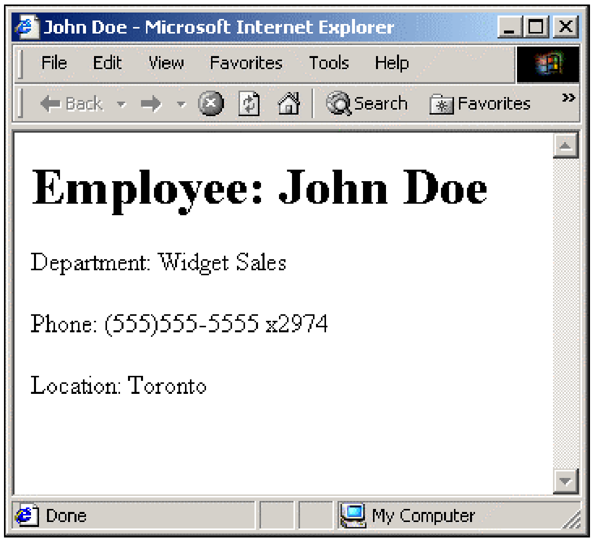
Salary: 62,000

Location: New York

7. Now change the `FullSecurity` attribute to "0", and the `<area>` element to "1", and re-run the transformation. The HTML will be changed to this:

```
>msxsl employee.xml EmployeeToHTML.xsl
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>John Doe</title>
</head>
<body>
<h1>Employee: John Doe</h1>
<p>Department: Widget Sales</p>
<p>Phone: (555)555-5555 x2974</p>
<p>Location: Toronto</p>
</body>
</html>
```

which looks like this in a browser:


John Doe - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Done
My Computer

Employee: John Doe

Department: Widget Sales

Phone: (555)555-5555 x2974

Location: Toronto

<xsl:for-each>

In many cases there is some specific processing that we want to do for a number of nodes in the source tree. For example, in the Try It Out section that demonstrated `<xsl:element>`, we had processing that we needed to do for every `<name>` element.

So far we've been handling this by creating a new template to do that processing, and then inserting an `<xsl:apply-templates>` element from where we wanted that template instantiated. But there is an alternative way to do this kind of processing, using `<xsl:for-each>`:

```
<xsl:for-each select="XPath expression">
```

The contents of `<xsl:for-each>` form a template-within-a-template; it is instantiated for any node matching the XPath expression in the `select` attribute. For example, consider the following:

```
<xsl:for-each select="name">
  This is a name element.
</xsl:for-each>
```

This "template" will be instantiated for every `<name>` element that is a child of the context node. In this case, all the template does is output the text "`This is a name element.`"

Because `<xsl:for-each>` is a template, it also changes the context node, as does a regular template. For example, consider the following XML:

```
<names>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
  <name>
    <first>Jane</first>
    <last>Smith</last>
  </name>
</names>
```

We could create HTML paragraphs for each `<first>` element, using a template like this:

```
<xsl:template match="names">
  <xsl:for-each select="name">
    <P><xsl:value-of select="first"/></P>
  </xsl:for-each>
</xsl:template>
```

When the template is instantiated, the context node is changed to the `<names>` element. So our `<xsl:for-each>` element need only specify `<name>` in the `select` attribute, as it's a child of this context node. Then, inside the `<xsl:for-each>`, the context node is changed again, to `<name>`. So the `select` attribute of the `<xsl:value-of>` only needs to specify the `<first>` element, as it's a child of this new context node.

Try It Out – <xsl:for-each> in Action

To demonstrate `<xsl:for-each>`, let's rewrite an earlier stylesheet, which transformed an attributes-only document to an elements-only document. The original stylesheet was this (which we saved earlier as `AttsToElems.xsl`):

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <people>
    <xsl:apply-templates select="people/name"/>
  </people>
</xsl:template>

<xsl:template match="name">
  <name>
    <xsl:apply-templates select="@*"/>
  </name>
</xsl:template>

<xsl:template match="@*">
  <xsl:element name="{name()}"><xsl:value-of select="."/></xsl:element>
</xsl:template>
</xsl:stylesheet>
```

1. And here it is again, with the changes highlighted. Save this as `AttsToElems2.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <people>
    <xsl:for-each select="people/name">
      <name>
        <xsl:apply-templates select="@*"/>
      </name>
    </xsl:for-each>
  </people>
</xsl:template>

<xsl:template match="@*">
  <xsl:element name="{name()}"><xsl:value-of select="."/></xsl:element>
</xsl:template>
</xsl:stylesheet>
```

2. Running this stylesheet through MSXSL produces the same results as the previous version:

```
>msxsl PeopleAtts.xml AttsToElems2.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
<name>
<first>Franklin</first>
<middle>D.</middle>
<last>Roosevelt</last>
</name>
<name>
<first>Alfred</first>
<middle>E.</middle>
<last>Neuman</last>
</name>
<name>
<first>John</first>
<middle>Q.</middle>
<last>Public</last>
</name>
<name>
<first>Jane</first>
<middle/>
<last>Doe</last>
</name>
</people>
```

How It Works

This stylesheet seems a bit simpler than the first one; it is much more straightforward and maintainable. We now only have two templates, instead of three, since the template that was used for `<name>` elements has now been moved up into the first template.

If we look closely at the `<xsl:for-each>`, which is highlighted in the code, we might notice that the contents are exactly the same as the template we replaced.

Remember, for all intents and purposes `<xsl:for-each>` *is* a template. The only difference between using `<xsl:for-each>` and `<xsl:template>` is that `<xsl:for-each>` can be inserted into other templates, whereas `<xsl:template>` must stand on its own.

Copying Sections of the Source Tree to the Result Tree

In many cases the result tree from our XSLT stylesheets will be very similar to the source tree. Perhaps there will even be large sections that are exactly the same. XSLT provides a couple of elements that you can use to copy sections of the source tree directly to the result tree, for these occasions.

<xsl:copy-of>

The `<xsl:copy-of>` element allows us to take sections of the source tree and copy them to the result tree. This is much easier than having to create all of the elements/attributes manually, and then copying the values using `<xsl:value-of>`, especially if we don't know ahead of time what the source tree will look like. The syntax is as shown:

```
<xsl:copy-of select="XPath expression"/>
```

The element is quite easy to use. The `select` attribute simply specifies an XPath expression pointing to the node or node-set required, and that node or node-set is inserted directly into the result tree, along with any attributes or child elements.

Try It Out – <xsl:copy-of> in Action

As a simple example of `<xsl:copy-of>`, let's re-examine our `employee.xml` document. We've already seen how to use the `FullSecurity` attribute to specify when to include the information from our `<salary>` element when creating HTML output. But what if we have another application that needs to use the raw XML, and we can't trust it to do the right thing with our security? It would be better to create an XSLT stylesheet which will remove `<salary>`, under the right conditions, before the other application even gets to touch it. So let's create one.

1. If you don't still have it, recreate the original employee XML document, and save it as `employee.xml` (if you already have it, simply change the `FullSecurity` attribute's value back to "1", and the `<area>` element's text node back to "3"):

```
<?xml version="1.0"?>
<employee FullSecurity="1">
```

```
    <name>John Doe</name>
    <department>Widget Sales</department>
    <phone>(555)555-5555<extension>2974</extension></phone>
    <salary>62,000</salary>
    <area>3</area>
  </employee>
```

2. Now enter the following, and save it as `EmployeeToXML.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <xsl:choose>
    <xsl:when test="number(employee/@FullSecurity)">
      <xsl:copy-of select="/"/>
    </xsl:when>
    <xsl:otherwise>
      <employee FullSecurity="{employee/@FullSecurity}">
        <xsl:for-each select="employee/*[not(self::salary)]">
          <xsl:copy-of select="."/>
        </xsl:for-each>
      </employee>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

3. When `FullSecurity` is "1", as above, the output is as follows:

```
>msxsl employee.xml EmployeeToXML.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="1">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555)555-5555<extension>2974</extension></phone>
  <salary>62,000</salary>
  <area>3</area>
</employee>
```

4. Now change `FullSecurity` to "`0`", and resave `employee.xml`. The new result is as follows:

```
>msxsl employee.xml EmployeeToXML.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="0">
<name>John Doe</name>
<department>Widget Sales</department>
<phone>(555)555-5555<extension>2974</extension></phone>
<area>3</area>
</employee>
```

How It Works

The first thing our template does is to check the `FullSecurity` attribute. If it's `true`, we call `<xsl:copy-of>` with the `select` attribute set to "`/`". This copies the entire document root, including all of its children and attributes, to the result tree.

However, if the attribute is `false`, we need to copy all of the nodes except for the `<salary>` element to the result tree. This is done using the `<xsl:for-each>` element, to loop through all of the child elements of the `<employee>` element that are not the `<salary>` element. (If you look at the `select` attribute, it selects all children of the `<employee>` element, using "`employee/*`". But it then further filters that, to select only the ones that aren't `<salary>`, using "`[not(self::salary)]`".)

> *The `not()` XPath function is a special Boolean function, that returns the reverse of whatever is passed to it. That is, if the expression would have evaluated to `true`, then `not()` will return `false`, and vice versa. In this case, we're creating an XPath expression that will create a node-set, containing all of the nodes which are* not *a `<salary>` node.*
>
> *There is also an XPath function called `true()`, which always returns `true`, and one called `false()`, which always returns `false`. These functions are most often useful when debugging stylesheets.*

This is one of the few places where the `self` axis is used, and it might seem a little strange. What we are saying is "if there is not a `<salary>` element in the `self` axis", and since the only node which is ever in the `self` axis is the context node, we're really checking this node's own name! Instead, we could have written that XPath as "`[not(local-name(.) = 'salary']`", which would have had the same effect. However, XSLT programmers use the construct we used in our stylesheet quite often, so it's good to see it in action, so that you won't be confused if you ever see it in a real-life stylesheet.

Each of these nodes is then copied to the result tree using `<xsl:copy-of>`.

<xsl:copy>

Similar to `<xsl:copy-of>` is the `<xsl:copy>` element. It allows much more flexibility when copying sections of the source tree to the result tree:

```
<xsl:copy use-attribute-sets="att set names">
```

Instead of providing a `select` attribute, to indicate which section of the source tree to copy, `<xsl:copy>` simply copies the context node. And, unlike the `<xsl:copy-of>` construct, children and attributes of the context node are not automatically copied to the result tree. However, the contents of the `<xsl:copy>` element provide a template where you specify the attributes and children of the node which will go into the result tree. For example, if we have our good old `<name>` XML:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

we could create a template that looked like this:

```
<xsl:template match="name">
  <xsl:copy/>
</xsl:template>
```

The output of this would be:

```
<name/>
```

since none of the children would be copied. However, we could modify the template to look like this:

```
<xsl:template match="name">
  <xsl:copy>
    <blah><xsl:value-of select="."/></blah>
  </xsl:copy>
</xsl:template>
```

in which case the output would look like this:

```
<name><blah>
 John
 Fitzgerald Johansen
 Doe
</blah></name>
```

Remember, when you use "." *for the* `select` *attribute in* `<xsl:value-of>`*, you get the contents of the context node* plus *any descendants.*

What we have done in the second example is to copy the context node, which in this case is the `<name>` element, created a child element called `<blah>`, and populated it with the text from `<name>` and its children.

Try It Out – <xsl:copy> in Action

Let's redo the style sheet from the last Try It Out, using `<xsl:copy>` instead of `<xsl:copy-of>`, and using a fancy tactic called **recursion**.

Recursion is a methodology whereby a function can call itself, which can then call itself again, etc. Of course, in the case of XSLT, instead of creating recursive *functions*, we would create recursive *templates.* These recursive templates are used quite a lot in XSLT, so it would be a good idea to start learning about them.

1. Since the best way to understand recursion is to see it in action, let's create the main template for this stylesheet:

```
<xsl:template match="/ | * | @*">
  <xsl:copy>
    <xsl:apply-templates select="* | @* | text()"/>
  </xsl:copy>
</xsl:template>
```

This template applies to the document root, as well as any elements, and any attributes. When an XSL processor starts going through the source tree, it will match the document root against this template. It will then be copied to the result tree, from the `<xsl:copy>`, and `<xsl:apply-templates>` will be called.

The XSL processor will then look through the source tree for further nodes, and when it finds any element or attribute, it will match against this template again. The new node will also be copied to the result tree, `<xsl:apply-templates>` will be called again, and any elements and attributes which are found will once more match against this template.

The template will keep getting called until it processes a node that doesn't have any child elements or attributes. Then it will return back to the template which called it and find further nodes to process, until it has processed all elements and attributes. (Because of the built-in template for text nodes we mentioned earlier, the XSLT engine will automatically add the text nodes' contents to the result tree.)

2. As we can see, if we were to just put this one template in our stylesheet, all elements and attributes, along with their associated text, would be copied to the result tree, as-is, every time. However, we need to keep that `<salary>` element from being copied when `FullSecurity` is `false`. In order to do that, we can create a second template, which will match against this specific case:

```
<xsl:template match="salary[not(number(parent::employee/@FullSecurity))]">
  <!--do not process this node-->
</xsl:template>
```

To explain that `match` attribute, the template matches any `<salary>` element, if the `FullSecurity` attribute of the `<employee>` parent element is `false`. And, since the template doesn't do anything, the node is not processed. We could also have accomplished this by simply using an empty element, like this:

```
<xsl:template match="employee[not(number(@FullSecurity))]/salary"/>
```

However, with the comment included in our version things are clearer.

In the case where `FullSecurity` is `false`, the `<salary>` element matches both templates. However, one of the rules we mentioned earlier is that the most explicit match wins. In this case, our first template matches against any element, whereas the second matches against this specific element, so the second one wins.

3. Putting it all together produces the following stylesheet, which you should save as `EmployeeToXML2.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" encoding="UTF-8"/>

<xsl:template match="/ | * | @*">
  <xsl:copy>
    <xsl:apply-templates select="* | @* | text()"/>
```

```
</xsl:copy>
</xsl:template>

<xsl:template match="salary[not(number(parent::employee/@FullSecurity))]">
  <!--do not process this node-->
</xsl:template>
</xsl:stylesheet>
```

4. Make sure the `FullSecurity` attribute in your XML is "`1`", and then run it through MSXSL. The results should look like this:

```
>msxsl employee.xml EmployeeToXML2.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="1">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555)555-5555<extension>2974</extension></phone>
  <salary>62,000</salary>
  <area>3</area>
</employee>
```

5. Now change `FullSecurity` to "`0`" and re-run it. The results will look like this:

```
>msxsl employee.xml EmployeeToXML2.xsl
<?xml version="1.0" encoding="utf-8"?>
<employee FullSecurity="0">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555)555-5555<extension>2974</extension></phone>

  <area>3</area>
</employee>
```

Sorting the Result Tree

Sorting in XSLT is accomplished by adding one or more `<xsl:sort>` children to an `<xsl:apply-templates>` element, or an `<xsl:for-each>` element:

```
<xsl:sort select="XPath expression"
    lang="lang"
    data-type="text or number"
    order="ascending or descending"
    case-order="upper-first or lower-first"/>
```

The `select` attribute chooses the element/attribute/etc. by which you want the XSL processor to sort. If more than one `<xsl:sort>` child is added, then the output is sorted first by the element/attribute/etc. in the first `<xsl:sort>`, and if there are any duplicates they are sorted by the element/attribute/etc. in the second, and so on.

If the XSL processor goes through all of the `<xsl:sort>` elements, and there are still two or more items with identical results, they are inserted into the result tree in the order in which they appear in the source tree.

The `data-type` attribute specifies whether the data you are sorting is numeric or textual in nature. For example, consider the numbers 1, 10, 5, and 11. If we sort these with `data-type` specified as "`text`", the result will be 1, 10, 11, 5. But if we sort numerically, with `data-type` specified as "`number`", the result will be 1, 5, 10, 11. The default is "`text`".

The `order` attribute specifies whether the result should be sorted in ascending or descending order. The default is ascending.

The `case-order` attribute specifies whether uppercase letters should come first, or lowercase letters should come first. For example, if `case-order` is "`upper-first`", then "A B a b" would be sorted as "A a B b", and if `case-order` is "`lower-first`", it would be sorted as "a A b B". The default value for `case-order` depends on the `lang` attribute, which specifies the language this document is in. When `lang` is set to "`en`", the default `case-order` is `upper-first`.

Try It Out – <xsl:sort> in Action

To demonstrate sorting with `<xsl:sort>`, let's sort our simple names XML.

1. If you don't still have it, recreate `PeopleElems.xml`, using the following XML:

```
<?xml version="1.0"?>
<people>
  <name>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <name>
    <first>Franklin</first>
    <middle>D.</middle>
    <last>Roosevelt</last>
  </name>
  <name>
    <first>Alfred</first>
    <middle>E.</middle>
    <last>Neuman</last>
  </name>
  <name>
    <first>John</first>
    <middle>Q.</middle>
    <last>Public</last>
  </name>
  <name>
    <first>Jane</first>
    <middle></middle>
    <last>Doe</last>
  </name>
</people>
```

2. The following XSLT stylesheet will take that XML and sort it, first by last name, then by first. Save it as `sort.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">
  <people>
    <xsl:for-each select="people/name">
      <xsl:sort select="last"/>
      <xsl:sort select="first"/>
      <xsl:copy-of select="."/>
    </xsl:for-each>
  </people>
</xsl:template>
</xsl:stylesheet>
```

Since ascending order is the default, we didn't bother to specify it.

3. Running this through MSXSL produces the following:

```
>msxsl PeopleElems.xml sort.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
   <first>Jane</first>
   <middle></middle>
   <last>Doe</last>
 </name>
<name>
   <first>John</first>
   <middle>Fitzgerald Johansen</middle>
   <last>Doe</last>
 </name>
<name>
   <first>Alfred</first>
   <middle>E.</middle>
   <last>Neuman</last>
 </name>
<name>
   <first>John</first>
   <middle>Q.</middle>
   <last>Public</last>
 </name>
<name>
   <first>Franklin</first>
   <middle>D.</middle>
   <last>Roosevelt</last>
 </name>
</people>
```

Modes

The `<xsl:template>` and `<xsl:apply-templates>` elements both have a `mode` attribute, which we have conveniently ignored up until now. What does the `mode` attribute do?

Modes are handy when you need to process the same section of XML more than once. It allows you to create these templates and only call the one you want.

Using modes is trivial. When you create the template, you specify the `mode` attribute, giving it a name to describe this mode. To call the template, use `<xsl:apply-templates>` as before, but with the addition of the `mode` attribute, specifying the same mode. Now, in addition to matching the templates against the source tree, the XSL processor will also make sure that the only templates called are ones which are part of this mode. For example, if we have the following XSLT:

```
<xsl:apply-templates select="name" mode="TOC"/>
```

and the following templates:

```
<xsl:template match="name" mode="TOC"/>
<xsl:template match="name" mode="body"/>
<xsl:template match="name"/>
```

only the first template will be instantiated, even though all of these templates match against the same section of the source tree.

Keep in mind that the built-in templates also apply when modes are being used. So, along with the default template that we saw earlier,

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

there is also a built-in template which works with modes, similar to

```
<xsl:template match="*|/" mode="mode">
  <xsl:apply-templates mode="mode"/>
</xsl:template>
```

where "mode" refers to the current mode. In other words, if an XSLT processor is currently working in "TOC" mode, and comes across an element that has no templates defined for it, the processor will apply a template like the following:

```
<xsl:template match="*|/" mode="TOC">
  <xsl:apply-templates mode="TOC"/>
</xsl:template>
```

Try It Out – Modes in Action

One common example given for modes is transforming an XML document to a web page. A template running under one mode can be used for a table of contents, and a template matching against the same section of the document can be run under another mode for the actual body of the document.

To demonstrate this, we'll use some XML to represent a chapter from one of the most brilliant XML books ever written.

1. Type in the following, and save it as `chapter.xml`:

```
<?xml version="1.0"?>
<chapter>
  <title>XSLT</title>
  <section>What is XSL?
    <paragraph><important>Extensible Stylesheet Language</important>, as the name
implies, is an XML-based language used to create
<important>stylesheets</important>.  An XSL engine uses these stylesheets
to...</paragraph>
    <paragraph>There are actually two...</paragraph>
  </section>
  <section>Why is XSLT So Important for E-Commerce?
    <paragraph>To get an idea of the power of XSLT...</paragraph>
    <paragraph>Unfortunately, the chances are...</paragraph>
  </section>
</chapter>
```

This is a bit shorter than the text of the actual chapter, but it's enough to demonstrate what we're doing. We want to take this and transform it to HTML, with a table of contents we can use to navigate to any section.

2. Our first step is to set up the template to match against the document root, which will output the main structure of the HTML file:

```
<xsl:template match="/">
  <html>
  <head><title><xsl:value-of select="chapter/title"/></title></head>
  <body>
  <h1><xsl:value-of select="chapter/title"/></h1>

  <h2>Table of Contents</h2>
  <ol>
    <xsl:apply-templates select="chapter/section" mode="TOC"/>
  </ol>

  <xsl:apply-templates select="chapter/section" mode="body"/>

  </body>
  </html>
</xsl:template>
```

The two important pieces of code are the ones where we call `<xsl:apply-templates>` to process the main body of our document. The same portion of the source tree will be processed twice by two separate templates, once for each mode.

3. Next we need to create those two templates. The first will output our table of contents:

```
<xsl:template match="section" mode="TOC">
    <li><a href="{concat('#section', position())}">
        <xsl:value-of select="text()"/></a>
    </li>
</xsl:template>
```

We're using two new XPath functions here: `position()` and `concat()`. The `position()` function returns a node's numeric position within a node-set; that is, for the first node in a node-set, `position()` would return "1", and for the second "2". The `concat()` function takes two strings, and concatenates them together (that is, combines them into one string). In this case, we're taking the string "`#section`", and adding it to the value returned from the `position()` function, to create strings similar to `"#section1`", "`#section2`", etc.

The next effect is that we are creating an HTML list item for each section in the body, using the `position()` function to make sure that every section is uniquely identified.

4. The second template will output the main body of the document:

```
<xsl:template match="section" mode="body">
  <a name="{concat('section', position())}"><h2>
     <xsl:value-of select="text()"/></h2>
  </a>
  <xsl:apply-templates/>
</xsl:template>
```

It matches against the `<section>` elements again, but works in "`body`" mode. This template creates an `<h2>` heading, with the title of the section, and wraps it in an `<a>` to identify it, so that the links from the table of contents will work. Then `<xsl:apply-templates>` is called, to take care of the rest of the work in creating this section.

5. The two following templates are very simple ones, which just transform the `<paragraph>` elements to HTML `<p>` elements, and the `<important>` elements to HTML `<b>` elements:

```
<xsl:template match="paragraph">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="important">
  <b><xsl:apply-templates/></b>
</xsl:template>
```

6. And finally, we come to the last template in the stylesheet, which matches against any PCDATA which is a direct child of a `<section>` element; that is, its title. This is done so that these titles won't make it into the body of the section itself.

The problem is those default templates, which all XSL processors implement. Remember that one of these default templates matches against any text that isn't matched against any other templates, and outputs that text to the result tree. This means that when we call `<xsl:apply-templates>` from within the body template, we would get the PCDATA which is a direct child of a `<section>` element inserted into the result tree as well, because of this default template. The template to hide these titles is as follows:

```
<xsl:template match="/chapter/section/text()"><!--do nothing--></xsl:template>
```

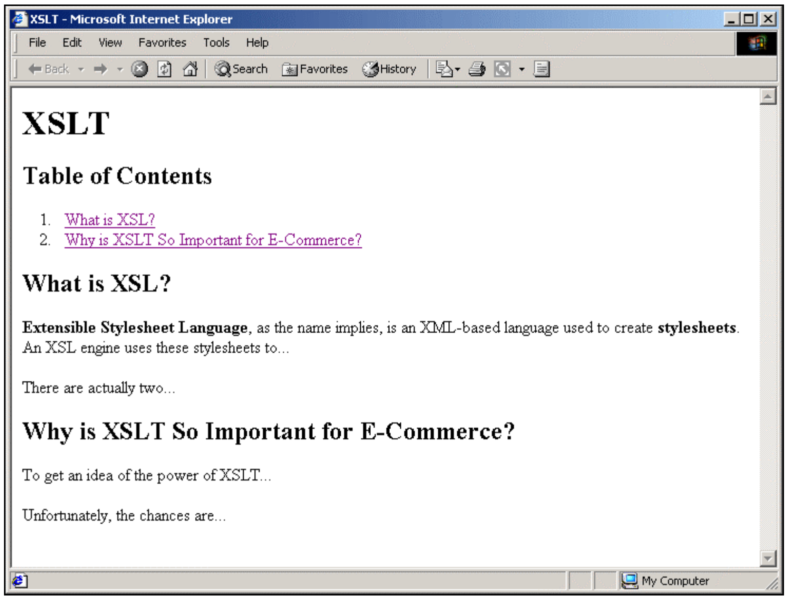
7. Putting all of this together produces the following stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" encoding="UTF-8"/>

<xsl:template match="/">
  <html>
  <head><title><xsl:value-of select="chapter/title"/></title></head>
  <body>
  <h1><xsl:value-of select="chapter/title"/></h1>

  <h2>Table of Contents</h2>
  <ol>
    <xsl:apply-templates select="chapter/section" mode="TOC"/>
  </ol>

  <xsl:apply-templates select="chapter/section" mode="body"/>

  </body>
  </html>
</xsl:template>

<xsl:template match="section" mode="TOC">
    <li><a href="{concat('#section', position())}"><xsl:value-of
select="text()"/></a></li>
</xsl:template>

<xsl:template match="section" mode="body">
  <a name="{concat('section', position())}"><h2><xsl:value-of
select="text()"/></h2></a>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="paragraph">
  <p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="important">
  <b><xsl:apply-templates/></b>
</xsl:template>

<xsl:template match="/chapter/section/text()"><!--do nothing--></xsl:template>
</xsl:stylesheet>
```

8. Save this as `chapter.xsl`, and run it through MSXSL like this:

```
>msxsl chapter.xml chapter.xsl -o chapter.html
```

You will get an HTML file called `chapter.html`, which contains the information from this chapter. In a browser, it will look similar to this:


XSLT - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
My Computer

XSLT

Table of Contents

1. What is XSL?
2. Why is XSLT So Important for E-Commerce?

What is XSL?

Extensible Stylesheet Language, as the name implies, is an XML-based language used to create **stylesheets**. An XSL engine uses these stylesheets to...

There are actually two...

Why is XSLT So Important for E-Commerce?

To get an idea of the power of XSLT...

Unfortunately, the chances are...

Variables, Constants, and Named Templates

Anyone who has worked with programming languages is familiar with **variables** and **constants**. Variables are places in your program where you store information, while constants are values which are predetermined when the program is written, and can never be changed. For example, in Java we could create a variable to store a person's age and a constant to store the value of Pi like this:

```
int nAge = 30;
final float cfPI = 3.14;
```

The `nAge` variable can then be changed any time we need to in our application, whereas `cfPI` can never be changed, but will always be 3.14:

```
nAge = 42;  //this is allowed
cfPI = 42;  //this is not allowed
```

Constants can be useful in a variety of situations. For example, if you were designing an application to print out reports to a printer, you might need to know throughout the application how many lines per page you're outputting, or what fonts to use. You could just manually put the values in wherever you need them, or you could define constants to store the values, and then use those. The usefulness of constants would become immediately apparent the first time you needed to change one of those values and recompile your program: instead of searching through all of the code for your application, and manually replacing each occurrence of the value, you would simply need to change the value of your constant, in one place. XSLT also provides us with mechanisms for adding constants and variables to stylesheets, using `<xsl:variable>` and named templates.

<xsl:variable>

The `<xsl:variable>` element allows us to add simple constants to stylesheets. For example, we could define a variable for Pi like so:

```
<xsl:variable name="cfPI">3.14</xsl:variable>
```

This variable can then be accessed anywhere you would use a regular XPath expression, using a dollar sign followed by the variable name. For example:

```
<math pi="{$cfPI}"/>
```

or:

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

But wait, there's more! `<xsl:variable>` can also contain XML markup, and even XSLT elements! For example:

```
<xsl:variable name="space">
  <xsl:text> </xsl:text>
</xsl:variable>

<xsl:variable name="name">
  <name>
    <xsl:value-of select="/name/first"/>
    <xsl:copy-of select="$space"/>
    <xsl:value-of select="/name/last"/>
  </name>
</xsl:variable>

<!--this gets the value of the $name variable, including any XML markup-->
<xsl:copy-of select="$name"/>
```

Notice that `$name` is allowed to reference `$space`. One important note, however, is that variables are not allowed to reference themselves, and neither are circular references allowed. (These would both be the XSLT equivalent of infinite loops. An XSL engine should catch this for you, to prevent your machine from crashing.) For example, the following *are not allowed* in XSLT:

```
<!--variables are not allowed to reference themselves-->
<xsl:variable name="name">
  <name><xsl:value-of select="$name"/></name>
</xsl:variable>

<!--circular references are not allowed either-->
<xsl:variable name="A">
  <b><xsl:value-of select="$B"/></b>
</xsl:variable>

<xsl:variable name="B">
  <a><xsl:value-of select="$A"/></a>
</xsl:variable>
```

As an alternative syntax, `<xsl:variable>` can have a `select` attribute. In this case, the value of the variable is the result from the XPath expression supplied. When `<xsl:variable>` has a `select` attribute, it is not allowed to have any content. For example:

```
<xsl:variable name="name" select="/people/name"/>
```

In all cases, whether the variable contains straight text or gets its value from the source tree, the value of the variable is fixed. This means that you can't change the value of a variable after it has been declared. (Refer back to the "*XSLT Has No Side Effects*" section earlier for the reasons behind this.) Programmers who are used to imperative languages will have a much easier time if they don't think in terms of variables when programming with XSLT, but treat `<xsl:variable>` like a constant.

When working with variables, it's important to remember the **binding**. That is, *where* in our stylesheet the constant can be referenced. (In other programming languages, this is often called the *scope*.) Consider the following example:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" encoding="UTF-8"/>

  <xsl:variable name="age">30</xsl:variable>

  <xsl:template match="/">
    <xsl:variable name="name">Fred</xsl:variable>
    <xsl:value-of select="concat($name, ' is ', $age)"/>
  </xsl:template>
</xsl:stylesheet>
```

When run with any XML file, this produces the following:

```
>msxsl anyfile.xml scope.xsl
Fred is 30
```

The `$age` variable is called a **global** variable, because it is defined outside of any templates, which makes it accessible anywhere in the stylesheet. `$name`, on the other hand, is a **local** variable, because it is defined inside a template, and therefore only accessible inside that template. It is also possible to create variables in different bindings with the same name. For example:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:variable name="name">Fred</xsl:variable>

<xsl:template match="/">
  <xsl:variable name="name">Barney</xsl:variable>
  <xsl:value-of select="$name"/>
</xsl:template>
</xsl:stylesheet>
```

In that `<xsl:value-of>`, which `$name` are we selecting? The answer is that the variable with the most restrictive binding is used. In this instance, the `$name` of "`Fred`" is global, and the `$name` of "`Barney`" is local to this template, so the second one is used. Running this stylesheet in MSXML against any XML file produces the following:

```
>msxsl anyfile.xml binding.xsl
Barney
```

Named Templates

Variables can be a powerful tool in XSLT stylesheets, but sometimes we need a bit more flexibility. Perhaps we need the functionality of a variable, but we also need to change the results returned from the variable depending on the source tree, similar to a template.

The problem with templates, though, from what we have learned so far, is that they work by matching against the source tree, so we can't just explicitly call a template whenever we want it. Furthermore, if we had one particular template which we wanted to instantiate for a number of different nodes in the source tree, we would have to define separate, identical templates for each node we match against, or else create a complex XPath expression for the template's `match` attribute, saying "match against this node, or that node, or this other node…".

Luckily, XSLT introduced the concept of **named templates**, which makes both of the above statements untrue. We create them using the same `<xsl:template>` element we have been using, but instead of using the `match` attribute, we specify the `name` attribute. These templates are then called with the simple `<xsl:call-template>` element. For example, consider the following XSLT:

```
<xsl:template match="/">
  <xsl:for-each select="date">
    <xsl:call-template name="bold"/>
  </xsl:for-each>

  <xsl:for-each select="customer">
    <xsl:call-template name="bold"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="bold">
  <b><xsl:value-of select="."/></b>
</xsl:template>
```

We have a simple template, which just outputs the contents of the context node, wrapped in an HTML `<b>` element. But this template is called both for `<date>` elements and for `<customer>` elements in the source tree; in fact, the bold template in this case doesn't care *what* elements it is operating on.

If you wish, you can create templates with both the `match` and the `name` attributes; this will create a normal template which is instantiated through `<xsl:apply-templates>`, but that can also be called explicitly if desired.

Parameters

At times we may not be able to make our templates generic enough for every case. For example, consider a template for outputting a name into the result tree:

```
<xsl:template name="name">
  First name:  <xsl:value-of select="first"/>
  Last name:  <xsl:value-of select="last"/>
</xsl:template>
```

In this case, we're assuming that the first name will always be in a `<first>` element, and the last name will always be in a `<last>` element. But what if we have an input document with names represented in different places in different ways? It might be better if we could specify to the template which values to use for the first name, and which values to use for the last.

The `<xsl:param>` element allows us to give our templates parameters, just like we give functions parameters in other programming languages. We could rewrite the previous template like this:

```
<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First name:  <xsl:value-of select="$first"/>
  Last name:  <xsl:value-of select="$last"/>
</xsl:template>
```

Now we're getting the values from our parameters. But, since those parameters are just getting their information from the `<first>` and `<last>` elements like before, how have we bought ourselves any benefits? The answer is that the information put inside the `<xsl:param>` element is only the **default value** for that parameter. So if we call our `name` template, without giving it any parameters, it will use the values from the `<first>` and `<last>` elements; if we call it with parameters, it will use the values we give it, instead of the defaults.

We pass parameters to the templates using the `<xsl:with-param>` element.

Stylesheets can also have parameters, by specifying top-level `<xsl:param>` elements. However, the XSLT specification doesn't specify how those parameters should be passed to the XSL processor, so the mechanics will vary from processor to processor. (For command-line XSLT engines, this is typically done through command-line parameters.)

Try It Out – Parameters in Action

Let's put the above `name` template into an actual stylesheet, and see it in action.

1. For our XML, we'll use the following format, which you should save as `person.xml`:

```
<?xml version="1.0"?>
<person>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
</person>
```

2. And here is our stylesheet, which you should save as `person.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="/">
  <xsl:for-each select="person/name">
    <xsl:call-template name="name"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First:  <xsl:value-of select="$first"/>
  Last:  <xsl:value-of select="$last"/>
</xsl:template>
</xsl:stylesheet>
```

Because the first and last names are contained in the `<first>` and `<last>` elements, just like our named template is expecting, we don't need to supply it with any parameters.

3. Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl

First:  John
Last:  Doe
```

4. Now let's change our XML document slightly, and resave it:

```
<?xml version="1.0"?>
<person>
  <name>
    <given>John</given>
    <family>Doe</family>
  </name>
</person>
```

5. And also change our stylesheet just a bit, and resave it:

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>
<xsl:template match="/">
  <xsl:for-each select="/person/name">

    <xsl:call-template name="name">
      <xsl:with-param name="first">
        <xsl:value-of select="given"/>
      </xsl:with-param>
      <xsl:with-param name="last">
        <xsl:value-of select="family"/>
      </xsl:with-param>
    </xsl:call-template>

  </xsl:for-each>
</xsl:template>

<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First:  <xsl:value-of select="$first"/>
  Last:  <xsl:value-of select="$last"/>
</xsl:template>
</xsl:stylesheet>
```

Since the first and last names aren't where the template expects to find them anymore, we pass it the parameters instead.

6. Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl

First:  John
Last:  Doe
```

which, you may notice, is the same output as last time.

7. Finally, let's modify our stylesheet one last time:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="/">
  <xsl:for-each select="/person/name">
    <xsl:call-template name="name">
```

```
        <xsl:with-param name="first">Fred</xsl:with-param>
        <xsl:with-param name="last">Garvin</xsl:with-param>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>

  <xsl:template name="name">
    <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
    <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

    First:  <xsl:value-of select="$first"/>
    Last:  <xsl:value-of select="$last"/>
  </xsl:template>
  </xsl:stylesheet>
```

This is just to demonstrate that the values passed to the parameters don't have to come from the source tree at all; in this case, we just pass it the strings "`Fred`" and "`Garvin`", and it uses those for its values.

8. Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl

First:  Fred
Last:  Garvin
```

Summary

This chapter has introduced Extensible Stylesheet Language Transformations, XSLT, a powerful template-based language that can be used to transform XML documents into, well, almost anything. We've covered:

- What XSLT is, and why it is important when working with XML
- What XPath is, and why a language like this is so necessary when working with XSLT
- Most of the XSLT elements you'll ever need when creating your own stylesheets

In all, we've learned how to put XSLT to use to style our XML documents in a number of exciting ways. The next chapter introduces Document Type Definitions, or DTDs.