

3

XSLT Basics

In this chapter, we will build on Chapters 1 and 2 to provide you with enough information to start building useful XSLT stylesheets. I will introduce a number of the elements that make up the language, providing examples of their use. We will also look at a few of the functions built into the language and see how XSLT manages namespaces, whitespace and some other important issues.

To illustrate the concepts I introduce, we will work mainly with two documents, one that is textual in content, and one that is more data oriented. The former is a Shakespeare play (Hamlet), and the latter is a book catalog that could, for example, have been extracted from a relational database. Both documents are given in the code download for the chapter.

By the end of the chapter, you will:

- ☐ have a clearer picture of the processing model of XSLT
- ☐ know the difference between push and pull model stylesheets, and when to use each
- ☐ understand the use of the most important XSLT elements
- ☐ understand the use of a few of the built-in functions
- ☐ understand the basic rules of how XSLT copes when there are conflicts in the stylesheet
- ☐ know more about the built-in template rules and how to over ride them

XSLT Processing

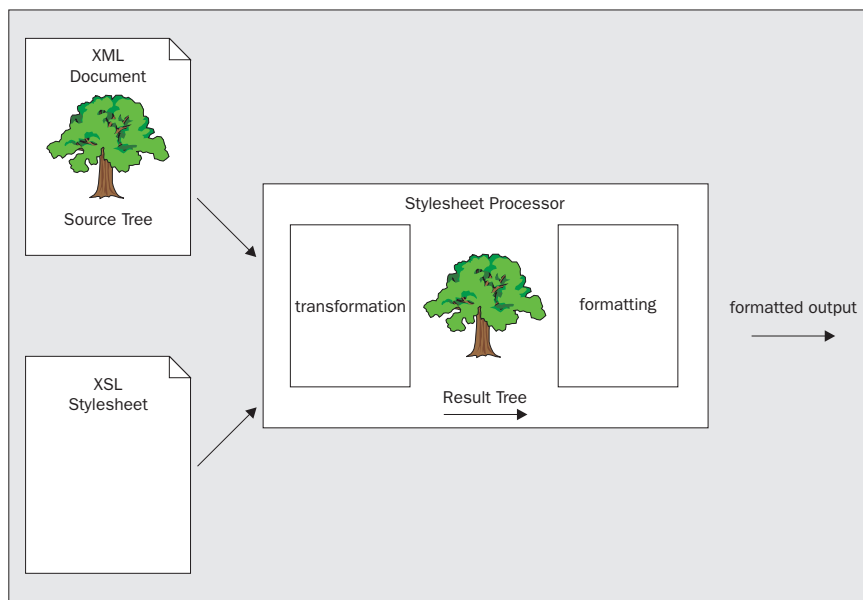
Before delving into the detail of XSLT elements and functions, let's start by looking in detail at how an XSLT processor, such as XT, Saxon or MSXML3, processes a document. We will look at the model from an abstract view – becoming an XSLT processor ourselves and working our way through a document and stylesheet. We'll then look at the two fundamental ways in which this model can be used.

More information on these processors can be found in Appendix E. Later in this chapter I will be mainly using XT to process XSLT stylesheets, but any of these processors can be used. XT is similar in use to Instant Saxon, which was introduced in Chapter 2.

The XSLT Processing Model

Although we often talk of an XSLT processor as something that turns one XML document into another (or into an HTML or text document), this is not strictly true. The specification actually talks in terms of a **source tree** (or **input tree**) and a **result tree**. There is therefore an assumption that, for example, if we are starting from a text document rather than an existing DOM tree, it has been turned into some sort of tree structure before the XSLT processor starts its work, and that the result tree will be used for further processing or serialized in some way to create another text document.

The model, including formatting, therefore looks like this:



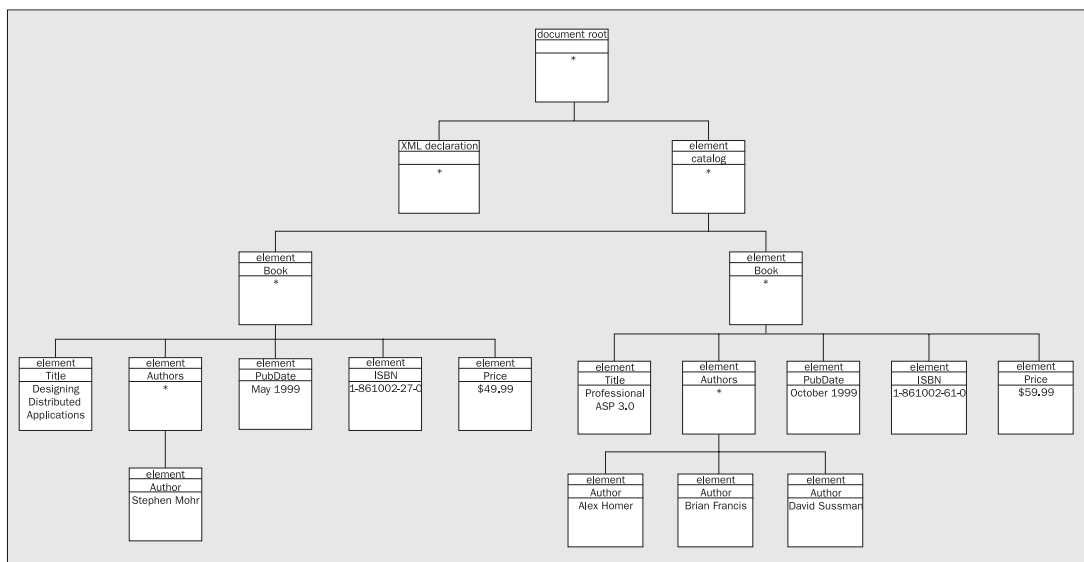
This concept is simple enough. But you will have read in Chapter 1 that XSLT is a declarative language and uses templates. How does this work in practice? Let's have a look at a simple XML document and stylesheet, and walk through the processing.

Processing a Document

Here is my XML document – it is the book catalog that you will be familiar with if you have read *Professional XML* (Wrox Press, ISBN 1-861003-11-0), although I have cut it down to just two books, removed some elements and renamed it `shortcatalog.xml`:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Catalog>
  <Book>
    <Title>Designing Distributed Applications</Title>
    <Authors>
      <Author>Stephen Mohr</Author>
    </Authors>
    <PubDate>May 1999</PubDate>
    <ISBN>1-861002-27-0</ISBN>
    <Price>$49.99</Price>
  </Book>
  <Book>
    <Title>Professional ASP 3.0</Title>
    <Authors>
      <Author>Alex Homer</Author>
      <Author>Brian Francis</Author>
      <Author>David Sussman</Author>
    </Authors>
    <PubDate>October 1999</PubDate>
    <ISBN>1-861002-61-0</ISBN>
    <Price>$59.99</Price>
  </Book>
</Catalog>
```

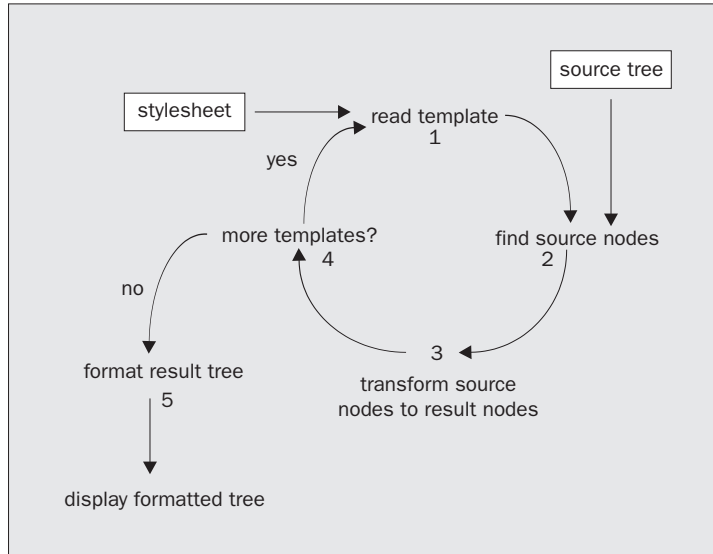
We'll look at the XSLT stylesheet we use to transform this document shortly, but let's now become an XSLT processor and see what happens. We already know that, as an XSLT processor, we cannot use the source XML, but need a tree representation based on the structure and content of the document. So here it is:



Each node is described by a block of three rectangles. In the top rectangle is the node type, with the node name in the rectangle below it. The bottom rectangle contains an asterisk if the node has element content, and the text if it has text content.

At the top of the tree is the **root node** or **document root**. Don't confuse this with the **root element** (or **document element**) familiar from XML. The document root is the base of the document, and has the document element (<Catalog>) as a child. It also has the XML declaration and any other top-level nodes (which might be comments or processing instructions) as children. The document element contains two child <Book> elements, and these hold the information about the books.

So now we have the tree structure, we can start to populate and process it. This is the processing model we will use:



Before XSL processing starts, both the source document and XSLT stylesheet must be loaded into the processor's memory. How this happens is dependent on the implementation. One option is that both are loaded as DOM documents under the control of a program. Another option is that the stylesheet is referenced by a processing instruction in the source XML document. IE5 can operate in this way, and will automatically load the stylesheet when the XML document is loaded.

And here is the XSLT stylesheet (TitleAndDate.xsl) we will use to process the shortcatalog.xml to get a new XML document listing just the titles of the books and their publication dates:

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

```

```

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

<xsl:template match="Book">
  <Book>
    <xsl:value-of select="Title"/>, <xsl:value-of select="PubDate"/>
  </Book>
</xsl:template>

</xsl:stylesheet>

```

Once the documents are in memory, we can start our processing. The XSL processor starts by reading the template for the document root from the stylesheet (step 1). Here is that template:

```

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

```

The first line indicates that it is a template, with a match attribute to indicate the node or nodes it is matching. The attribute value is an XPath expression, in this case just being the / to indicate the document root.

Working round the diagram, at step 2 we find the source node (strictly, the node-set, but here it will comprise a single node) in the source tree that the template matches. This will be the document root. The second line of the template moves us on to step 3 and indicates that we will execute whatever templates apply to the children of this node. The document root has two children – the XML declaration and the <Catalog> element.

Looking through the stylesheet, there is no template for the XML declaration (XSLT does not give us access to this node), but there is one for the <Catalog> element. Processing a document using XSL is a recursive process, and we are now back to step 1 with a new template. Here is the template:

```

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

```

This contains some text, which looks like another element called <Books>. As our diagram indicates, we will transform this into a result node at step 3. It also contains an <xsl:apply-templates/> instruction, so we will again look for templates to execute matching the child nodes.

The only children of the <Catalog> element are the two <Book> elements, so we will read the template for these elements and go round the circle again. Here is the template:

```

<xsl:template match="Book">
  <Book>
    <xsl:value-of select="Title"/>, <xsl:value-of select="PubDate"/>
  </Book>
</xsl:template>

```

This time, for each `<Book>` element we are creating a `<Book>` element in the result tree. Into this, we are placing the value of the `<Title>` element, then some literal text comprising a comma and a space, then the value of the `<PubDate>` element.

Note that the value of an element in XSLT is not the same as with the Document Object Model (DOM). With the DOM, the value of an element is always null, while in XSLT it is the text between the start and end tags.

At this point we stop since we have no more `<xsl:apply-templates/>` elements. This means that no other elements in the source document will get processed, but then that's what we wanted.

So how are we constructing the result tree? Let's work this one from the bottom up. When we execute the template for `<Book>`, we create the new `<Book>` element, and then replace the line:

```
<xsl:value-of select="Title"/>, <xsl:value-of select="PubDate"/>
```

with the result of evaluating the statements. For the first book, that will be:

```
Designing Distributed Applications, May 1999
```

So overall, our result node will look like:

```
<Book>Designing Distributed Applications, May 1999</Book>
```

Since we have two `<Book>` elements in the source tree, we will get two `<Book>` elements in the result tree:

```
<Book>Designing Distributed Applications, May 1999</Book>
<Book>Professional ASP 3.0, October 1999</Book>
```

Similarly, in the template for `<Catalog>`, we will replace the line:

```
<xsl:apply-templates/>
```

with the results generated by executing the instruction. This will put the two `<Book>` elements we have created inside a `<Books>` element. The result tree now looks like this:

```
<Books>
  <Book>Designing Distributed Applications, May 1999</Book>
  <Book>Professional ASP 3.0, October 1999</Book>
</Books>
```

I have added the line breaks and formatting to make the output look better.

Moving back to the first template we came across, the one for the document root, we can see that this adds no further content, so our output is exactly as I have just shown.

In our processing model, we now break out of our cycle and format the output (step 5). In this case, we have no formatting, so there is no further processing. Later in this book, we will see how we can format using a standard web-browser and HTML, or using the Formatting Objects part of the XSL specification (XSL-FO).

Note that the XSLT specification says that "... XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL." However, in the majority of cases, XSLT is used independently of XSL-FO, just as we are doing here and will do again when we produce HTML using XSLT. The specification acknowledges this with the statement "... XSLT is also designed to be used independently of XSL."

Using any of the processors described in Appendix E we can run the XSLT stylesheet with the XML. For example, if we now invoke XT with the command line:

```
xt shortcatalog.xml TitleAndDate.xsl TitleAndDate.xml
```

we produce a file `TitleAndDate.xml` with the content:

```
<?xml version="1.0" encoding="utf-8"?>
<Books>
  <Book>Designing Distributed Applications, May 1999</Book>
  <Book>Professional ASP 3.0, October 1999</Book>
</Books>
```

XT has put an XML declaration at the top, but otherwise it is exactly as we generated ourselves.

Push and Pull Models

In HTML, there is only one way of applying styles with an external stylesheet, and that is by using **Cascading Style Sheets (CSS)**. In this case, the structure of the output is usually determined by the source HTML, while the CSS stylesheet determines the appearance of each item within that structure. There are exceptions to this – some aspects of CSS2, such as tables and the use of absolute positioning, allow the stylesheet to control the structure of the output. Unfortunately, CSS2 is not well supported by browsers, and absolute positioning can give a display that is very dependent on the size of the browser window.

The CSS model can be referred to as a **push model**. The source document controls the format, while the stylesheet controls the appearance within that structure. An alternative model is a **pull model**, where the stylesheet provides the structure, and the XML document acts as a data source. As we will see in a moment, XSLT can support both push and pull models, each being appropriate in different circumstances.

The Push Model with CSS

This extract from a Shakespearean play (`HamletExtract.xml`) will be used in the following samples to illustrate the push model of stylesheet design:

```
<?xml version="1.0"?>
<EXTRACT>
  <ACT><TITLE>ACT I</TITLE>

  <SCENE><TITLE>SCENE I. Elsinore. A platform before the castle.</TITLE>
  <STAGEDIR>FRANCISCO at his post. Enter to him BERNARDO</STAGEDIR>

  <SPEECH>
    <SPEAKER>BERNARDO</SPEAKER>
    <LINE>Who's there?</LINE>
  </SPEECH>
```

```
<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>Nay, answer me: stand, and unfold yourself.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>Long live the king!</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>Bernardo?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>He.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>You come most carefully upon your hour.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>'Tis now struck twelve; get thee to bed, Francisco.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>For this relief much thanks: 'tis bitter cold,</LINE>
<LINE>And I am sick at heart.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>Have you had quiet guard?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>Not a mouse stirring.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>Well, good night.</LINE>
<LINE>If you do meet Horatio and Marcellus,</LINE>
<LINE>The rivals of my watch, bid them make haste.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>FRANCISCO</SPEAKER>
<LINE>I think I hear them. Stand, ho! Who's there?</LINE>
</SPEECH>
</SCENE>
</ACT>
</EXTRACT>
```


Jon Bosak has marked up all of Shakespeare's plays in XML. An index to them can be found at <http://www.andrew.cmu.edu/user/akj/shakespeare> and a zip file to download them all at <http://metalab.unc.edu/bosak/xml/eg/shaks200.zip>.

With a play such as this, we will generally want our display to follow the structure of the source. We can style the XML with CSS in just the same way as we would with HTML. All we have to do is associate styles with element names as we have here in `Hamlet.css`:

```
ACT TITLE {
    display:block;
    font-size:24pt;
    font-weight:bold;
    margin-bottom:12pt;
}

SCENE TITLE {
    display:block;
    font-size:16pt;
    margin-bottom:6pt;
}

STAGEDIR {
    display:block;
    font-style:italic;
    margin-top:6pt;
    margin-bottom:6pt;
}

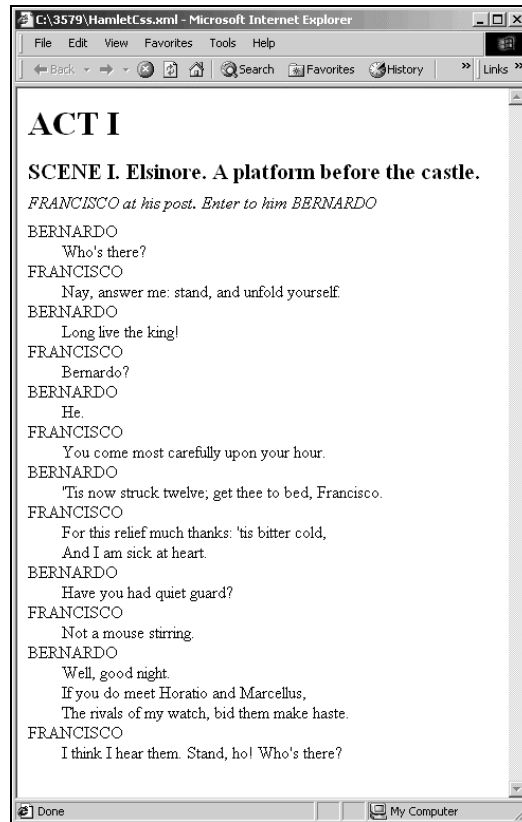
SPEAKER {
    display:block;
}

LINE {
    display:block;
    margin-left:2em;
}
```

If we add a line to reference the CSS stylesheet near the top of the file `HamletExtract.xml` to create `HamletCss.xml`:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="Hamlet.css"?>
<EXTRACT>
<ACT><TITLE>ACT I</TITLE>
```

then we can view the file in IE5 or Netscape 6. This is the result in IE5:



As you can see, the order of the text in the display is the same as that in the XML source, which is what we mean by a push model.

If you are used to using CSS with HTML, you might have noticed in particular the statement:

```
display:block;
```

within each style rule. This is used here to ensure that each of the elements starts on a new line. The alternatives are `display:inline` and `display:none`. You probably don't use these in HTML, as HTML contains elements such as `<DIV>` and `<P>` that are implicitly block elements, while others, such as `<I>` and ``, are implicitly inline elements. However, with XML, elements do not contain implicit display features like this, so the line is essential. I will not discuss the rest of the CSS here, since it is covered in more detail in Chapter 9.

The Push Model with XSLT

So how do we achieve the same sort of effect with XSLT? It is not difficult to create an identical display to that above by using XSLT to transform the XML into a document that combines HTML and CSS. We will be seeing that later in the book, but for now we will create a simple HTML page.

We can do this easily since both HTML, as an application of SGML, and XML, as a subset of SGML, use similar formats. The important thing is that our XSLT stylesheet is well-formed XML. My file, `Hamlet.xsl`, is one such example:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="EXTRACT">
    <HTML>
      <HEAD>
        <TITLE>Hamlet</TITLE>
      </HEAD>
      <BODY>
        <xsl:apply-templates/>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="ACT/TITLE">
    <H1><xsl:value-of select="." /></H1>
  </xsl:template>

  <xsl:template match="SCENE/TITLE">
    <H2><xsl:value-of select="." /></H2>
  </xsl:template>

  <xsl:template match="STAGEDIR">
    <P><I><xsl:value-of select="." /></I></P>
  </xsl:template>

  <xsl:template match="SPEAKER">
    <DIV><xsl:value-of select="." /></DIV>
  </xsl:template>

  <xsl:template match="LINE">
    <DIV><xsl:value-of select="." /></DIV>
  </xsl:template>

</xsl:stylesheet>
```

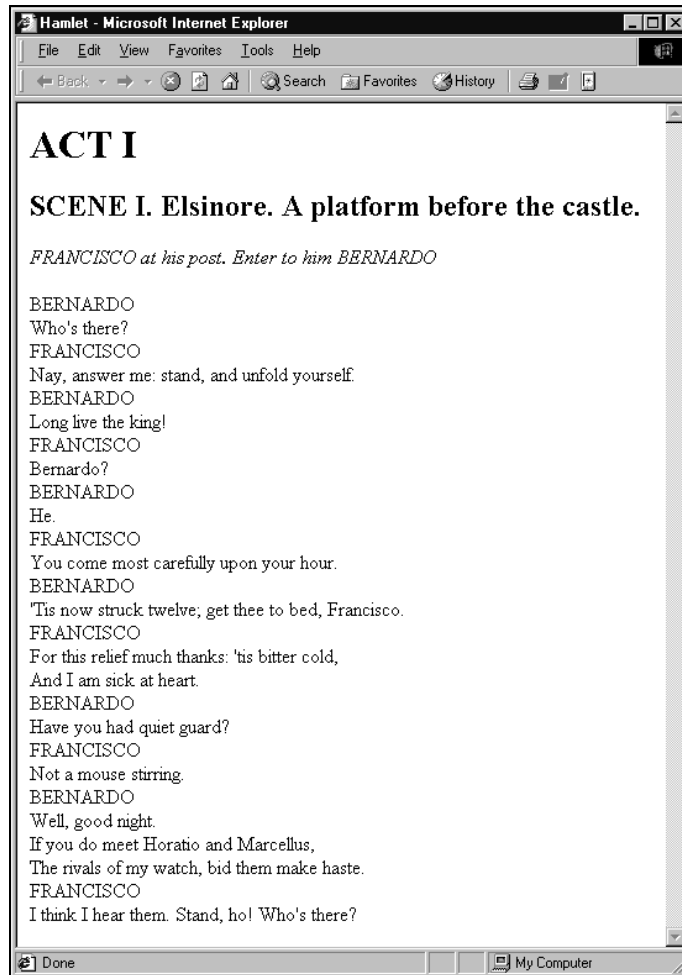
If we change the top of our XML document `HamletExtract.xml` to use the above transformation like this:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Hamlet.xsl"?>
<EXTRACT>
<ACT><TITLE>ACT I</TITLE>
```

and save it as `HamletXsl.xml`, we can view the result in IE5.

Note that to view this directly, you will need to have MSXML3 installed in replace mode (see Appendix E). Otherwise, you can just carry out the transformation using XT or Saxon, and view the resulting HTML.

This is the result in IE5 (this time, you cannot use Netscape since the processing instruction we have used is specific to IE):



and this is the HTML that is generated by XT – that is, using the command:

```
xt HamletExtract.xml Hamlet.xsl Hamlet.html
```

```
<HTML>
<HEAD>
<TITLE>Hamlet</TITLE>
</HEAD>
<BODY>
<H1>ACT I</H1>
```

```

<H2>SCENE I.  Elsinore.  A platform before the castle.</H2>
<P>
<I>FRANCISCO at his post. Enter to him BERNARDO</I>
</P>
<DIV>BERNARDO</DIV>
<DIV>Who's there?</DIV>

<DIV>FRANCISCO</DIV>
<DIV>Nay, answer me: stand, and unfold yourself.</DIV>

<DIV>BERNARDO</DIV>
<DIV>Long live the king!</DIV>

<DIV>FRANCISCO</DIV>
<DIV>Bernardo?</DIV>

<DIV>BERNARDO</DIV>
<DIV>He.</DIV>

<DIV>FRANCISCO</DIV>
<DIV>You come most carefully upon your hour.</DIV>

<DIV>BERNARDO</DIV>
<DIV>'Tis now struck twelve; get thee to bed, Francisco.</DIV>

<DIV>FRANCISCO</DIV>
<DIV>For this relief much thanks: 'tis bitter cold,</DIV>
<DIV>And I am sick at heart.</DIV>

<DIV>BERNARDO</DIV>
<DIV>Have you had quiet guard?</DIV>

<DIV>FRANCISCO</DIV>
<DIV>Not a mouse stirring.</DIV>

<DIV>BERNARDO</DIV>
<DIV>Well, good night.</DIV>
<DIV>If you do meet Horatio and Marcellus,</DIV>
<DIV>The rivals of my watch, bid them make haste.</DIV>

<DIV>FRANCISCO</DIV>
<DIV>I think I hear them. Stand, ho! Who's there?</DIV>

</BODY>
</HTML>

```

I have used `<DIV>` elements rather than, say, `<P>` elements, as they do not add vertical space. However, it is easy to change this in the XSLT stylesheet if you prefer a different appearance. In later chapters, we will use CSS to provide more control over the display format.

So how did this work? Let's work through again from the root node. As before, the template for the root node causes all templates for its children to be executed. In this case, we will execute the template for the `<EXTRACT>` element next. This simply builds the structure of our HTML page, then executes any templates matching its children such that results are inserted inside an HTML `<BODY>` element.

Most of the other templates are self-explanatory and build up the HTML page. But what happens when we look for a template that matches the `<SPEECH>` element? In fact, we have no template for this element, but we do have a template with `match="*|/"`. In XPath, the pipestem symbol (`|`) acts as an "or" and the asterisk means match any element. We will therefore execute this template for the `SPEECH` element. Our template reaches the children of this element (`<SPEAKER>` and `<LINE>`) because of the `<xsl:apply-templates/>` in the template. How do we know not to apply this template for other elements, such as `<LINE>`, which have their own templates? This is a matter for which templates have priority or precedence, topics we will cover later in the chapter.

As with the CSS example, this is clearly a push model of a stylesheet. As we come across an element in the source document, we execute its template. This type of stylesheet is typified by the presence of several short templates and the use of `<xsl:apply-templates/>`. Now we will look at a stylesheet that uses the pull model – something that is next to impossible with CSS.

The Pull Model with XSLT

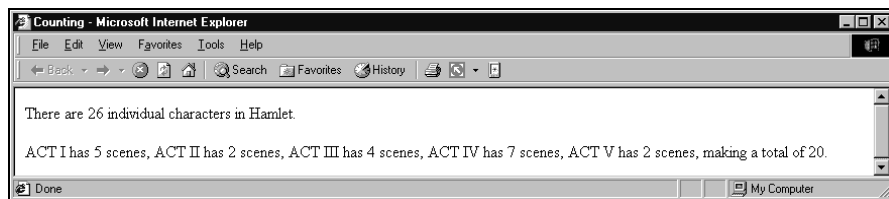
In a pull model stylesheet, the XML source document acts purely as a data source and its structure is largely irrelevant. The stylesheet itself provides the structure of the output document.

Let's look at a simple example based on the full source of the play, `Hamlet.xml`, which is provided in the code download for the book. Here is our stylesheet, `count.xsl`:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="PLAY">
    <HTML>
      <HEAD>
        <TITLE>Counting</TITLE>
      </HEAD>
      <BODY>
        <P>There are <xsl:value-of select="count(//PERSONA)"/> individual
          characters in Hamlet.</P>
        <P>
          <xsl:for-each select="ACT">
            <xsl:value-of select="TITLE"/> has <xsl:value-of
              select="count(SCENE)"/> scenes,
          </xsl:for-each>
          making a total of <xsl:value-of select="count(//SCENE)"/>.
        </P>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

This is the result of applying the stylesheet to the play (for example, by using XT to apply the stylesheet to `Hamlet.xml` and produce the HTML result):



This bears little resemblance to anything in the play. Instead, the stylesheet has provided the structure of the output, pulling in data as it is needed. You will also notice the use of the built-in `count()` function in this example. We will meet this later when we look at the elements and functions in detail.

The pull model is characterized by a few large templates and use of the `<xsl:value-of>` element so that the stylesheet controls the order of items in the output. Compare this to the push model, where we had more and smaller templates with the output largely following the structure of the XML source document.

I mentioned earlier that XSLT is often thought of as a declarative language. However, it also contains the flow control and looping instructions typical of a procedural language. Typically, a push model stylesheet emphasizes the declarative aspects of the language, while the pull model emphasizes the procedural aspects.

Of course, these definitions are not absolute. Most stylesheets will contain elements of the push and elements of the pull models. However, it is useful to keep the two models in mind as it can make your stylesheet development simpler.

We have now looked at how XSLT stylesheets are created and seen examples of both push and pull models. In doing this, we have come across several of the XSLT elements and their attributes, and the built-in `count()` function. We will look at these and other elements in more detail, shortly.

A Word About Namespaces

You can get a long way with XML without any knowledge of **namespaces**, but you won't get far with XSL! For a start, an XSL processor uses namespaces to indicate which elements to process itself and which to put directly into the result tree. Let's start with a quick review of namespaces in general, and then look at the three main uses of namespaces in XSL.

Because we can define our own element and attribute names in XML, there is always the chance that we will want to use the same name with different meanings in a document. If I am processing XML that is describing metalworking, I might want to create an element in my output called `<template>`. As we have seen, XSLT has an element with the same name, so I might try something like this:

```
<template match="pattern">
  <template><value-of select="."/></template>
</template>
```

The purpose of XML namespaces is to make our element names unique. In this case, it would disambiguate my two `<template>` elements, so that the XSL processor knows that it must process the first, but that the second should be copied directly to the output tree.

XML namespaces use a **Uniform Resource Identifier (URI)** to ensure uniqueness of names.

Why a URI? Simply that a key aspect of a URI is that it is unique. When we use the URI such as `http://www.wrox.com/namespaces`, a registration authority allocates the domain name `wrox.com` to Wrox Press and to nobody else. It is then up to Wrox to ensure that any URIs within this domain are unique.

So we can make our `template` elements unique by associating them with a namespace. One way to do this is with a **namespace prefix**, and that is what we have been doing with the XSLT namespace in our examples so far. We have associated the prefix `xsl` with the URI `http://www.w3.org/1999/XSL/Transform` using the reserved `xmlns` attribute within our stylesheet element:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Since the `stylesheet` element itself is part of the XSLT namespace, this also uses the prefix. As you can see, the prefix is separated from the element name by a colon.

In the terminology of the W3C recommendation (<http://www.w3.org/TR/REC-xml-names>), `xsl:stylesheet` is a **qualified name** (or **QName**), `xsl` is a **prefix** and `stylesheet` is a **local part**.

So we could modify our stylesheet to look like this:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

Now the XSL processor knows to process the elements starting `xsl:` and to place other elements directly into the result tree.

Let's also declare a namespace for our `<pattern>` and `<template>` elements:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:wrox="http://www.wrox.com/namespaces">

  <xsl:template match="wrox:pattern">
    <wrox:template><value-of select="."/></wrox:template>
  </xsl:template>
</xsl:stylesheet>
```

Now that we have done this, we can make either of these namespaces the default namespace for the document by removing the colon and prefix in the namespace declaration and the prefix and colon in element and attribute names that use that namespace. Note that we can do this with either namespace, so the following two documents are equivalent.

Here is the document with the Wrox namespace as default:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.wrox.com/namespaces">

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


This is the document with the XSLT namespace as default:

```
<stylesheet
  version="1.0"
  xmlns="http://www.w3.org/1999/XSL/Transform"
  xmlns:wrox="http://www.wrox.com/namespaces">

  <template match="wrox:pattern">
    <wrox:template><value-of select="."/></wrox:template>
  </template>
</stylesheet>
```

In the example above, we have used namespaces for three distinct purposes, although we used the Wrox namespace for two of these:

- ❑ to indicate which elements were part of the XSLT namespace and so should be processed by the XSL processor
- ❑ to match elements, such as the `<pattern>` element, in the source tree only if they belong to a certain namespace
- ❑ to ensure that elements in the result tree, such as `<template>`, belong to the namespace we want

Let's just look a little more at matching namespace-qualified nodes in the source tree. Take the case where we have a stylesheet that is operating on an XML Schema document. We might have a line in our stylesheet such as:

```
<xsl:apply-templates select="xsd:complexType"/>
```

If we have not specified the namespace we are using for the `xsd` prefix in our stylesheet, our XSLT processor will match on the complete name `xsd:complexType` (since the colon is a valid part of an XML element name). If we then have another schema using `xs` as a qualifier rather than `xsd`, or using the XML Schema namespace as the default, we will no longer get matches in our stylesheet. This is unlikely to be what we want. By specifying the XML Schema namespace in our stylesheet, we can be certain that we will match XML Schema elements whatever prefix they are using, or even if they have no prefix because they belong in the default namespace of our source document.

In general in this book, we will use the `xsl` prefix for XSLT elements and suitable qualifiers for other namespaces.

XSLT Elements

In this section, we will recap all the elements we have used so far, and meet several others. With these you will be able to create the vast majority of the XSLT stylesheets you might want. In the next chapter, we will meet more elements; some of these you will require less often, while others provide more advanced functionality.

We will be looking at the most frequently used aspects of these elements. For fuller descriptions refer either to the XSLT recommendation or to a source such as the *XSLT Programmer's Reference 2nd Edition* (ISBN 1-861005-06-7 from Wrox Press).

<xsl:stylesheet>

This is simply the container element for all other elements within an XSL stylesheet. In most cases, this means it will be the document element of a stylesheet document.

A stylesheet can be embedded within another document, in which case an `id` attribute of this element can be used to allow a reference to the stylesheet.

The `<xsl:stylesheet>` element must contain a `version` attribute, indicating the version of XSLT that is being used. Currently, this is always 1.0 or 1.1.

The most important change between version 1.0 and version 1.1 of XSLT is that the latter allows multiple output documents to be created from a single XML source document and stylesheet. The full list of changes is documented as an appendix to the XSLT 1.1 specification (<http://www.w3.org/TR/xslt11>), which is at working draft stage at the time of writing.

The element is also likely to contain several namespaces, as we saw above. Firstly, there will be the XSLT namespace itself to tell the XSL processor which elements to process and which to pass unchanged to the output tree. Then there might be a namespace for XSL formatting objects (which we will look at in Chapter 9), namespaces for elements and attributes we will be matching in the source document, and namespaces for elements we might be creating in the output document. For example, if we wanted to use a stylesheet to document an XML Schema document, producing HTML output, our `xsl:stylesheet` start tag might look like this:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns="http://www.w3.org/TR/REC-html40">
```

In this case, we have defined HTML as our default namespace, and used explicit qualifiers for the XSLT and XML Schema namespaces.

Other optional attributes of the `<xsl:stylesheet>` element relate to namespace prefixes in the result tree and extension elements (which we will meet later in the book).

<xsl:output>

This element is used to inform the XSL processor of the format of the result tree. Earlier we used the following example (`count.xml`) without using `<xsl:output>`:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="PLAY">
    <HTML>
      <HEAD>
        <TITLE>Counting</TITLE>
      </HEAD>
      <BODY>
```

```

    ...
  </BODY>
</HTML>
</xsl:template>

</xsl:stylesheet>

```

We used various HTML elements directly in our stylesheet, and these were copied directly to the result tree. Since the stylesheet is an XML document, the HTML included in it must itself be well-formed XML, and hence the HTML copied to the result tree will also be well-formed XML. This well-formed XML could meet the rules of the Extensible Hypertext Markup language, XHTML (see *Beginning XHTML*, ISBN 1-861003-43-9), but this is not essential to the operation of the stylesheet.

If the transformed result tree is being saved as a file, the last stage of XSLT processing will be to serialize the result tree. It seems reasonable to assume that this will also be well-formed XML. In most cases, this would not cause problems, but some HTML browsers (particularly older ones) have difficulty with constructs such as `<HR/>`, preferring just the opening tag `<HR>` without a closing tag for a horizontal rule. HTML also allows attributes without values (as in `<OPTION selected>`). Again, this is not well-formed XML, but some browsers will object to the alternative form of `<OPTION selected="selected">`. For this reason, alternative forms of serialization are supported through the `<xsl:output>` element. When, as in the example above, the element is omitted, the serialized output will obey rules we will look at later. In this case, it would be XML.

This element has an optional `method` attribute that specifies the form that the serialization should take. The three possible values are `xml`, `html` and `text`. The `xml` option is simple enough – the serialized output will be well-formed XML. The `html` option handles the cases shown above by converting the tags to the more normal HTML styles of `<HR>` and `<OPTION selected>`, and the `text` method provides a pure text output, removing all tags and converting entity and character references to their text equivalents.

Let's take a simple stylesheet, `output.xsl`, that creates some HTML, and try the different forms of `<xsl:output>`. Since we now know more about the use of namespaces in XSL, we will also declare the unqualified names copied to the output to be in the HTML namespace.

```

<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40">

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

  <xsl:template match="/">
    <HTML>
    <HEAD><TITLE>Testing the xsl:output element</TITLE></HEAD>
    <BODY>
      <P>
        This is a simple stylesheet to show the effect of the xsl:output element.
        There is an &lt;HR/> element after this line.
      </P>
      <HR/>
      <SELECT>
        <OPTION value="1">First option</OPTION>

```

```

        <OPTION selected="selected" value="2">Second (selected) option</OPTION>
        <OPTION value="3">Third option</OPTION>
    </SELECT>
</BODY>
</HTML>
</xsl:template>

</xsl:stylesheet>

```

In general in this chapter, we will leave out the HTML namespace declaration for simplicity.

We have specified our output method as `html`. We have also specified that we want the result indented, by putting in the attribute `indent="yes"`. Although the XSLT recommendation does not specify what action the XSL processor should take as a result of this, with some processors it can make the result more readable.

What source XML shall we apply this to? The answer is any! This is the ultimate pull model stylesheet – it totally ignores the input XML, creating its result tree based only on the content of the stylesheet. Since we have some XML we used earlier, we can run the stylesheet using:

`xt hamlet.xml output.xsl output.htm`

We are not really interested in what the resulting HTML looks like when rendered. What we want to see is the HTML code produced (`output.htm`):

```

<HTML xmlns="http://www.w3.org/TR/REC-html40">
<HEAD>
<TITLE>Testing the xsl:output element</TITLE>
</HEAD>
<BODY>
<P>
    This is a simple stylesheet to show the effect of the xsl:output element.
    There is an &lt;HR/&gt; element after this line.
</P>
<HR>
<SELECT><OPTION value="1">First option</OPTION><OPTION selected value="2">Second
(selected) option</OPTION><OPTION value="3">Third option</OPTION></SELECT>
</BODY>
</HTML>

```

Note that this is "traditional" HTML. If we want to create XHTML, which is well-formed XML, we should use an output method of `xml` since the changes made by the `html` method do not create a well-formed XML document. We would also have to make other changes to the stylesheet, such as changing all element names to lower case.

Now, changing just one line of `output.xsl` will switch the output method to `xml`:

```

<xsl:output method="xml" indent="yes"/>

```

Save the file as `output2.xml`. The result of using this stylesheet is that the `<HR>` element is changed to the empty tag syntax and the `selected` attribute of the `<OPTION>` element is given its attribute value:

```
<?xml version="1.0" encoding="utf-8"?>
<HTML xmlns="http://www.w3.org/TR/REC-html40">
<HEAD>
<TITLE>Testing the xsl:output element</TITLE>
</HEAD>
<BODY>
<P>
    This is a simple stylesheet to show the effect of the xsl:output element.
    There is an &lt;HR/&gt; element after this line.
</P>
<HR/>
<SELECT>
<OPTION value="1">First option</OPTION>
<OPTION selected="selected" value="2">Second (selected) option</OPTION>
<OPTION value="3">Third option</OPTION>
</SELECT>
</BODY>
</HTML>
```

Note that, although I have specified my output as XML, this does not mean I am producing XHTML. That is up to me to control, by obeying XHTML rules such as using lower case for all tag names.

With XT, the `indent` attribute did not affect the HTML output, but it improved the layout of the XML version. With any XSL processor, it is best to experiment to see the difference this attribute makes.

Finally, let's make one more change to our `output.xml`:

```
<xsl:output method="text" indent="yes"/>
```

Save the file as `output3.xml`. Our result when using XT now looks like this:

```
Testing the xsl:output element
    This is a simple stylesheet to show the effect of the xsl:output element.
    There is an <HR/> element after this line.
    First optionSecond (selected) optionThird option
```

All the tags have now been removed, and the entity references `<` and `>` have now been replaced by their corresponding characters. This is clearly important if we are, for example, using our stylesheet to create a comma-separated file from an XML input document. Another attribute of `<xsl:output>` that helps under these circumstances is the `encoding` attribute. This allows us to specify a character set such as `iso-8859-1` for our output. Any character outside this set will cause an error to be reported.

Earlier, we were using XSLT without the `<xsl:output>` element and creating well-formed XML. This is normally the default, but if the result tree meets all of the following three criteria, the serialized output will be HTML by default:

- ❑ the root node has at least one element child
- ❑ the expanded-name of the first element child of the root node of the result tree has local part `html` (in any combination of upper and lower case) and a null namespace URI
- ❑ any text nodes preceding the first element child of the root node of the result tree contain only whitespace characters

Other attributes of `<xsl:output>` provide much more control over the output. They can:

- ❑ define the version of XML or HTML being created
- ❑ control aspects of the XML declaration
- ❑ indicate the SYSTEM and PUBLIC identifiers of the DOCTYPE
- ❑ control the MIME type of the output
- ❑ control how CDATA sections are handled

These are described in detail in the XSLT recommendation and in reference books such as the *XSLT Programmer's Reference*.

<xsl:template>

We have just looked at `<xsl:output>`, which is known as a **top-level** element as it can only occur as a child of the `<xsl:stylesheet>` element. We'll now look at `<xsl:template>`, which is another top-level element.

The `<xsl:template>` element contains the information required to produce a node in the result tree. We have seen several examples in the stylesheets we have been producing.

In the examples we have used so far, we have employed a `match` attribute, for example:

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

The attribute value is an XPath expression that will tell us which nodes in the source tree will cause this template to execute.

When using XPath expressions, there is always the possibility that several templates will match a single element. For example, if you look back to the listing of `HamletExtract.xml`, you will see that both `acts` and `scenes` have titles. In our previous stylesheet `Hamlet.xsl`, we had the templates:

```
<xsl:template match="ACT/TITLE">
  <H1><xsl:value-of select="." /></H1>
</xsl:template>

<xsl:template match="SCENE/TITLE">
  <H2><xsl:value-of select="." /></H2>
</xsl:template>
```

What if we had another template:

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

Which template would be executed when our `<xsl:apply-templates>` finds a scene's title?

One thing we can do is add a `priority` attribute to the `<template>` element:

```
<xsl:template match="TITLE" priority="1">
  <P><xsl:value-of select="." /></P>
</xsl:template>
```

This will allow the stylesheet to select the highest priority template, and only execute that. The value of the `priority` attribute can be any positive or negative integer or real number: the higher the number, the higher the priority. The default priority is between -0.5 and 0.5, so this new template will always be matched for any `<TITLE>` element, and the more specific `ACT/TITLE` and `SCENE/TITLE` templates will never be matched.

This is just one method of resolving conflicts when there are multiple templates matching a pattern. Later, in the section *Template Match Conflicts*, we will look at the others and the rules that determine in general which template is instantiated.

The `<xsl:template>` element has two further optional attributes. One is `name`, which is used with the `<xsl:call-template>` element, and the other is `mode`, which provides additional flexibility in the use of templates. We will meet both these attributes in Chapters 4 and 5.

<xsl:apply-templates>

Along with the `<xsl:template>` and `<xsl:value-of>` elements, this is the workhorse of the XSLT world. It is the element that controls which templates are used at any point while building the result tree.

It can have two attributes, of which we have already met the `select` attribute. This simply takes as its value an XPath expression that controls which elements in the source tree will be processed. This pattern can be simple, like most of those we have been using, or much more complicated, using any of the rules of XPath. Note that the `select` attribute is optional. If it is missing, all child nodes will be processed.

The second attribute is the `mode` attribute. Since this relates to the `mode` attribute in `<xsl:template>`, we will again be leaving this to Chapter 4.

As well as its attributes, `<xsl:apply-templates>` can have two sub-elements. The first is `<xsl:sort>`, which alters the order in which selected nodes are processed, and the second is `<xsl:with-param>`, which provides a method of passing parameters to templates. We will look at `<xsl:sort>` shortly, but leave `<xsl:with-param>` to Chapter 4.

<xsl:value-of>

This element is always contained within a template and simply writes the text node (or nodes) of the element pointed to by the XPath expression identified in its `select` attribute to the result tree.

The `select` attribute works in much the same way as it does in `<xsl:apply-templates>`, taking an XPath expression as its value. However, with `<xsl:apply-templates>`, if there are several matches, the relevant templates are instantiated multiple times. With `<xsl:value-of>`, only the first instance is used. Also, unlike with `<xsl:apply-templates>`, in this case the attribute is mandatory.

The value of this expression is always written to the result tree as a string, so some conversion might need to take place first. A number is converted to the string representation of that number. If a node contains sub-nodes, the value is the concatenation of the values of all the sub-nodes. So, for example, for our document `HamletExtract.xml`, we could include within the `<SCENE>` template a line:

```
<xsl:value-of select="SPEECH[3]" />
```

This will return a value that is the concatenation of the values of the `<SPEAKER>` and `<LINE>` elements of the third `<SPEECH>`. This is the speech that will be selected:

```
<SPEECH>
<SPEAKER>BERNARDO</SPEAKER>
<LINE>Long live the king!</LINE>
</SPEECH>
```

So the text returned will be `BERNARDOLong Live the king!`.

Finally, if the result is a Boolean, it will be converted to one of the strings `true` or `false`.

<xsl:copy> and <xsl:copy-of>

These two elements copy information from a source node directly to the result tree. `<xsl:copy>` performs what is known as a **shallow** copy. This means that it copies only the node and any namespace – not its descendants or attributes. `<xsl:copy-of>` performs a **deep** copy, copying not only the node and any namespace, but all its attributes and descendants.

`<xsl:copy-of>` is useful for copying sections of the XML source tree unchanged to the result tree, while `<xsl:copy>` gives more control of what will be copied. For example, we might want to list some key data from `Hamlet.xml`. We could do this with the following simple stylesheet (`Playkey.xsl`):

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="text()" />

  <xsl:template match="PLAY">
    <xsl:copy>
```



```

    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

<xsl:template match="TITLE | PERSONAE">
  <xsl:copy-of select="." />
</xsl:template>

</xsl:stylesheet>

```

The result of this is to copy the element `<PLAY>` (but not its descendants) to the result tree as the document element, then list the title of the play, the *Dramatis Personae*, and the title of each Act and Scene. Note that we have used the line:

```
<xsl:template match="text()" />
```

to ensure that the built-in templates do not cause extraneous text to be output. The result when this is applied to `Hamlet.xml` looks like this:

```

<?xml version="1.0" encoding="utf-8"?>
<PLAY>
<TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
<PERSONAE>
<TITLE>Dramatis Personae</TITLE>

<PERSONA>CLAUDIUS, king of Denmark. </PERSONA>
<PERSONA>HAMLET, son to the late, and nephew to the present king.</PERSONA>
<PERSONA>POLONIUS, lord chamberlain. </PERSONA>
<PERSONA>HORATIO, friend to Hamlet.</PERSONA>
<PERSONA>LAERTES, son to Polonius.</PERSONA>
<PERSONA>LUCIANUS, nephew to the king.</PERSONA>

<PGROUP>
<PERSONA>VOLTIMAND</PERSONA>
<PERSONA>CORNELIUS</PERSONA>
<PERSONA>ROSENCRANTZ</PERSONA>
<PERSONA>GUILDENSTERN</PERSONA>
<PERSONA>OSRIC</PERSONA>
<GRPDESCR>courtiers.</GRPDESCR>
</PGROUP>

<PERSONA>A Gentleman</PERSONA>
<PERSONA>A Priest. </PERSONA>

<PGROUP>
<PERSONA>MARCELLUS</PERSONA>
<PERSONA>BERNARDO</PERSONA>
<GRPDESCR>officers.</GRPDESCR>
</PGROUP>

<PERSONA>FRANCISCO, a soldier.</PERSONA>
<PERSONA>REYNALDO, servant to Polonius.</PERSONA>
<PERSONA>Players.</PERSONA>

```

```

<PERSONA>Two Clowns, grave-diggers.</PERSONA>
<PERSONA>FORTINBRAS, prince of Norway. </PERSONA>
<PERSONA>A Captain.</PERSONA>
<PERSONA>English Ambassadors. </PERSONA>
<PERSONA>GERTRUDE, queen of Denmark, and mother to Hamlet. </PERSONA>
<PERSONA>OPHELIA, daughter to Polonius.</PERSONA>
<PERSONA>Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other
Attendants.</PERSONA>
<PERSONA>Ghost of Hamlet's Father. </PERSONA>
</PERSONAE>
<TITLE>ACT I</TITLE>
<TITLE>SCENE I.  Elsinore. A platform before the castle.</TITLE>
<TITLE>SCENE II.  A room of state in the castle.</TITLE>
<TITLE>SCENE III.  A room in Polonius' house.</TITLE>
<TITLE>SCENE IV.  The platform.</TITLE>
<TITLE>SCENE V.  Another part of the platform.</TITLE>
<TITLE>ACT II</TITLE>
<TITLE>SCENE I.  A room in POLONIUS' house.</TITLE>
<TITLE>SCENE II.  A room in the castle.</TITLE>
<TITLE>ACT III</TITLE>
<TITLE>SCENE I.  A room in the castle.</TITLE>
<TITLE>SCENE II.  A hall in the castle.</TITLE>
<TITLE>SCENE III.  A room in the castle.</TITLE>
<TITLE>SCENE IV.  The Queen's closet.</TITLE>
<TITLE>ACT IV</TITLE>
<TITLE>SCENE I.  A room in the castle.</TITLE>
<TITLE>SCENE II.  Another room in the castle.</TITLE>
<TITLE>SCENE III.  Another room in the castle.</TITLE>
<TITLE>SCENE IV.  A plain in Denmark.</TITLE>
<TITLE>SCENE V.  Elsinore. A room in the castle.</TITLE>
<TITLE>SCENE VI.  Another room in the castle.</TITLE>
<TITLE>SCENE VII.  Another room in the castle.</TITLE>
<TITLE>ACT V</TITLE>
<TITLE>SCENE I.  A churchyard.</TITLE>
<TITLE>SCENE II.  A hall in the castle.</TITLE>
</PLAY>

```

Of course, we could equally well use the same stylesheet for other plays marked up in the same format.

This example shows a common use of `<xsl:copy-of>`, but perhaps a rather contrived example for `<xsl:copy>`, since we could equally well have used a different template for the `<PLAY>` element:

```

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

```

This would also have allowed us to use a different element name in the result tree from that in the source tree. So let's look at another use of `<xsl:copy>`.

We said earlier that `<xsl:copy>` gives us more control over the copying operation than `<xsl:copy-of>`. We used this just now to control which descendants of the `<PLAY>` element we would copy. Another use is if we have a source tree that is mainly XHTML, but with some additional elements from a different namespace. Much of the web site at <http://www.alphaxml.com> is written in XHTML. However, there is a glossary application that is used to explain technical terms. An example of the source code that might be used on this site is:

```
<p>The <em>Extensible Markup Language</em> provides a universal mechanism for
marking up data on the web. Unlike <g:term>HTML</g:term>, which is a language
based around displaying data in a web browser, <g:term>XML</g:term> puts no
constraints on the purpose for which the data will be used, but merely describes
the structure of the data. XML can therefore be used (and is used) for any
application that involves the transfer of data across the web for either display
or computation purposes.</p>
```

Here, the `<p>` and `` elements are from the XHTML namespace, which is used as the default for the document. The `<term>` element is from a different namespace for the glossary, using the prefix `g`.

When we process this, we want to copy the XHTML elements unchanged, but carry out some additional processing on the `<g:term>` elements. `<xsl:copy>` lets us do this. This is an extract from the template used on that site (reformatted for display):

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

<xsl:template match="g:term">
  <a target="_blank" class="glossary">
    <xsl:attribute name="href">
      glossary/glossary.asp?glossary=AlphaGlossary.xml&term=
      <xsl:value-of select="."/>
    </xsl:attribute>
    <xsl:value-of select="."/>
  </a>
</xsl:template>
```

In this example, the first template will be run for all elements, attributes and text nodes apart from the `<g:term>` element (for which the second template takes priority). This will copy the node to the result tree and apply templates for its children. If one of those children is a `<g:term>` element, the second template will be run. This simply creates an anchor (`<a>`) element in the result tree with an `href` attribute whose value depends on the enclosed text. For the term "HTML", the result will be:

```
<a
  target="_blank"
  class="glossary"
  href="glossary/glossary.asp?glossary=AlphaGlossary.xml&term=HTML">
  HTML
</a>
```

Control Flow Elements

The following five elements allow us to control execution within a template in a manner analogous to procedural languages. These are `<xsl:if>`, `<xsl:choose>`, `<xsl:when>`, `<xsl:otherwise>`, and `<xsl:for-each>`. The first of these gives a simple `if... then... construct`. The next three provide the equivalent of `if... then... else...` and the `switch` statement in many languages, and the last provides looping.

<xsl:if>

This element is used within a template purely to make execution of the enclosed statements conditional on the result of a test. It has a mandatory `test` attribute, which contains an expression that will return a Boolean result. The enclosed statements will be executed if the result of the test is `true`.

The test expression may involve the use of XSLT functions. We shall cover a couple of the functions included in XSLT at the end of this chapter, but their use in the following code is self-explanatory.

Common uses for `<xsl:if>` are testing for error conditions, or treating the first or last elements of a collection differently from the others. For example, the following template (`ListCharacters.xml`) lists all the characters in Hamlet, placing a comma and space after all but the last:

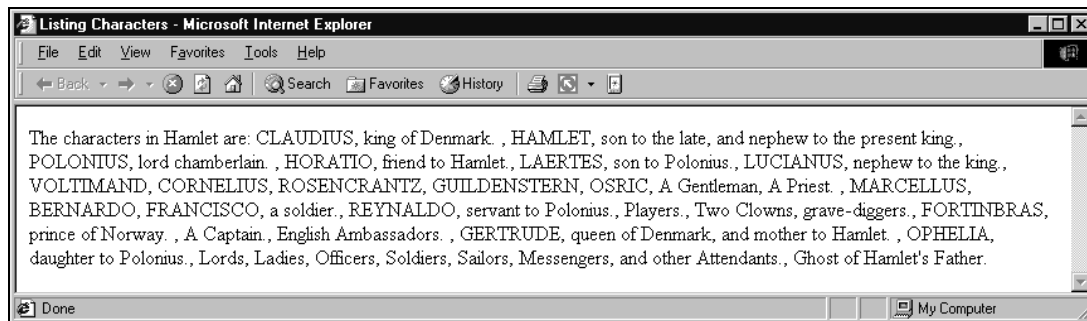
```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="PLAY">
  <HTML xmlns="http://www.w3.org/TR/REC-html40">
    <HEAD>
      <TITLE>Listing Characters</TITLE>
    </HEAD>
    <BODY>
      <P>
        The characters in Hamlet are:
        <xsl:for-each select="//PERSONA">
          <xsl:value-of select="."/>
          <xsl:if test="position() != last()">, </xsl:if>
        </xsl:for-each>
      </P>
    </BODY>
  </HTML>
</xsl:template>

</xsl:stylesheet>
```

And this is the result after being applied to `Hamlet.xml`:



Note that this element only allows an `if... then...`; if we want an `else...`, we must use `<xsl:choose>`, which we will look at next.

<xsl:choose>, <xsl:when>, and <xsl:otherwise>

These elements provide the equivalent of a switch statement, and can therefore also be used to provide an if... then... else... construct.

Here is an example of them in use in `HamletWithLines.xsl`, which is modified from a previous stylesheet (`Hamlet.xsl`):

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

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

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

  <xsl:template match="EXTRACT">
    <HTML xmlns="http://www.w3.org/TR/REC-html40">
      <HEAD>
        <TITLE>Hamlet</TITLE>
      </HEAD>
      <BODY>
        <xsl:apply-templates/>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="ACT/TITLE"><H1><xsl:value-of select="."/></H1></xsl:template>

  <xsl:template match="SCENE/TITLE"><H2><xsl:value-of
select="."/></H2></xsl:template>

  <xsl:template match="STAGEDIR"><P><I><xsl:value-of
select="."/></I></P></xsl:template>

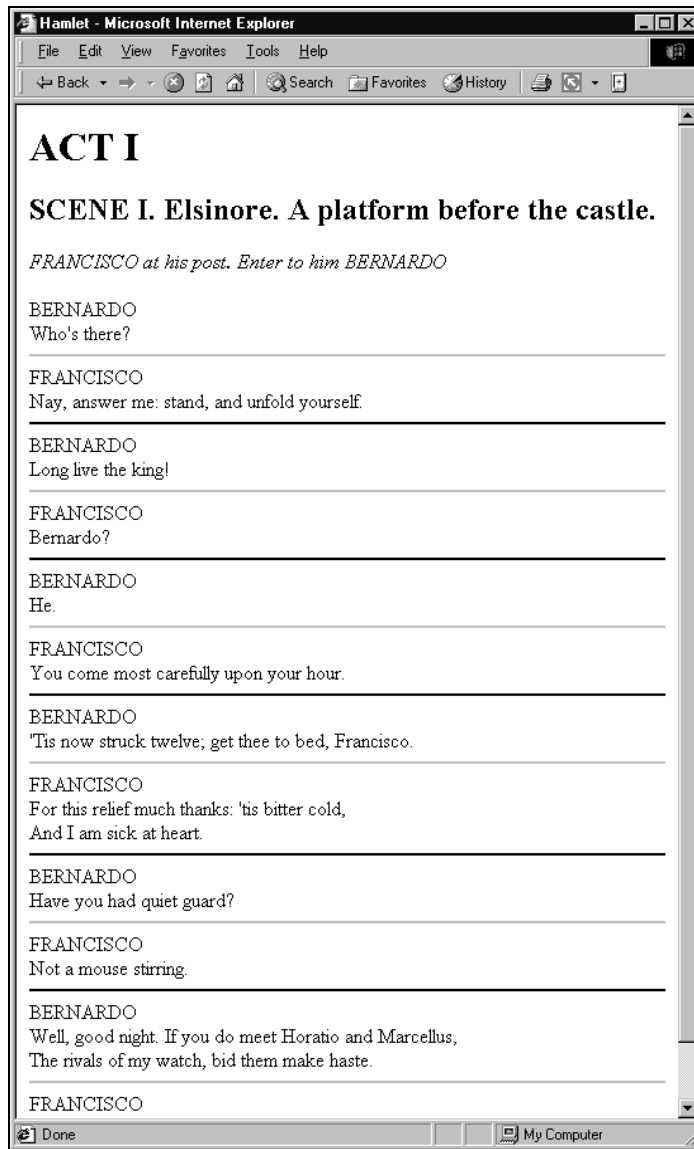
  <xsl:template match="SPEAKER"><DIV><xsl:value-of select="."/></DIV></xsl:template>

  <xsl:template match="LINE[position()=last()]">
    <DIV>
      <xsl:value-of select="."/>
      <xsl:choose>
        <xsl:when test="../SPEAKER='BERNARDO'">
          <HR style="color:silver"/>
        </xsl:when>
        <xsl:when test="../SPEAKER='FRANCISCO'">
          <HR style="color:black"/>
        </xsl:when>
        <xsl:otherwise> <!-- this is the trap for unrecognized speakers -->
          <DIV style="color:silver">
            !! oops, I don't know this speaker !!
          </DIV>
        </xsl:otherwise>
      </xsl:choose>
    </DIV>
  </xsl:template>

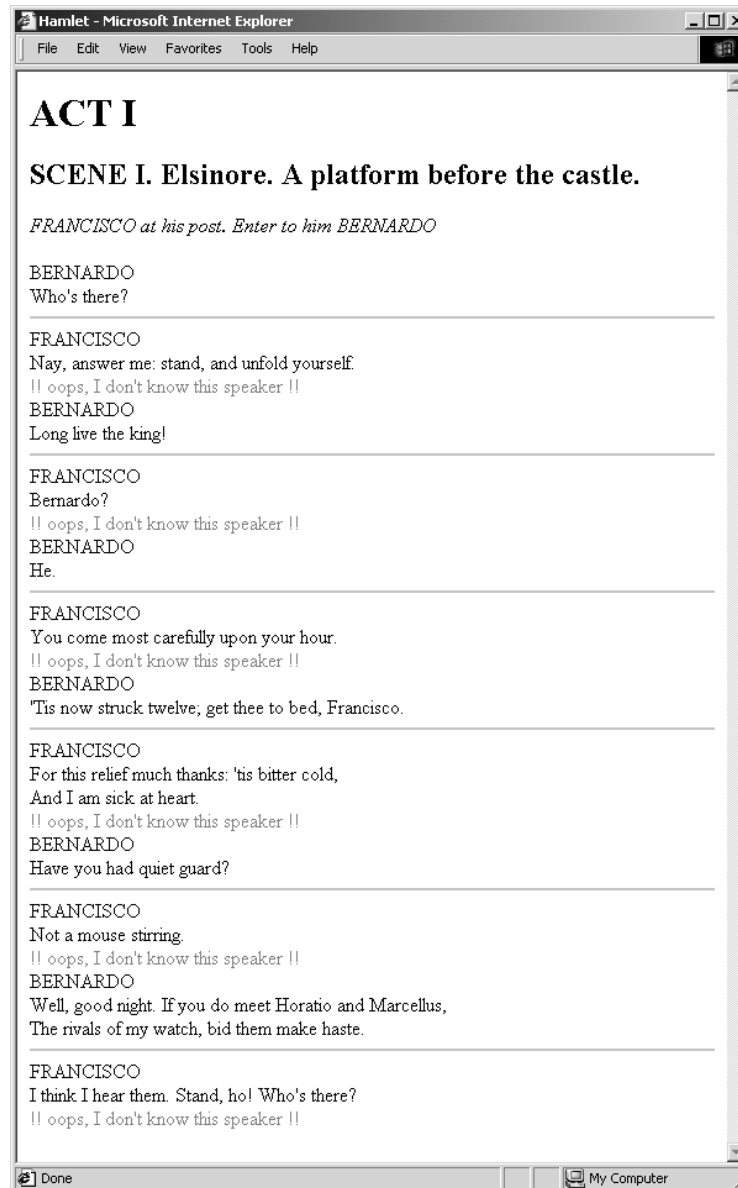
  <xsl:template match="LINE"><xsl:value-of select="."/></xsl:template>

</xsl:stylesheet>
```

I have added the `<xsl:output>` element and HTML namespace that we did not know about earlier. But the important part is the new template, which simply puts a different shade of horizontal rule under the last line of each speech, depending on the speaker. This is how it looks when applied to `HamletExtract.xml`:



In my first attempt at this stylesheet, I misspelled the name "FRANCISCO" as "FRANSISCO" in the new template. This was the result:



As you can see, the `<xsl:otherwise>` can be used to trap errors, as it has done here.

<xsl:for-each>

The last of the control flow elements, <xsl:for-each>, has already been seen in action when we were counting the number of scenes in each act. Here is the stylesheet (count.xsl) that we used:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="PLAY">
  <HTML>
    <HEAD>
      <TITLE>Counting</TITLE>
    </HEAD>
    <BODY>
      <P>There are <xsl:value-of select="count(//PERSONA)"/> individual
        characters in Hamlet.</P>
      <P>
        <xsl:for-each select="ACT">
          <xsl:value-of select="TITLE"/> has
          <xsl:value-of select="count(SCENE)"/> scenes,
        </xsl:for-each>
        making a total of <xsl:value-of select="count(//SCENE)"/>.
      </P>
    </BODY>
  </HTML>
</xsl:template>

</xsl:stylesheet>
```

The <xsl:for-each> statement effectively allows us to embed a template inside another. Instead of the <xsl:for-each> statement, we could have used <xsl:apply-templates>, and made the contents of the <xsl:for-each> a separate template:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="PLAY">
  <HTML>
    <HEAD>
      <TITLE>Counting</TITLE>
    </HEAD>
    <BODY>
      <P>There are <xsl:value-of select="count(//PERSONA)"/> individual
        characters in Hamlet.</P>
      <P>
        <xsl:apply-templates select="ACT"/>
        making a total of <xsl:value-of select="count(//SCENE)"/>.
      </P>
    </BODY>
  </HTML>
</xsl:template>

  <xsl:template match="ACT">
    <xsl:value-of select="TITLE"/> has
    <xsl:value-of select="count(SCENE)"/> scenes,
  </xsl:template>

</xsl:stylesheet>
```


So when should you use `<xsl:for-each>` and when should you use `<xsl:apply-templates>`? Often, the same template will be executed as a result of several XPath expression matches. In this case, using `<xsl:apply-templates>` provides re-usability in the same way that a function call does in a procedural language. On other occasions, it is largely a matter of style, and you should use whichever makes your stylesheet easier to understand. As we saw earlier, a push model stylesheet tends to use `<xsl:apply-templates>` most of the time to execute specific template rules as elements are found in the source tree, while a pull model is more likely to have a single large template with `<xsl:for-each>` statements to execute the embedded rule at a specific point in the stylesheet.

XSLT has the concept of a **context** node, which is important when using relative XPath expressions. In the stylesheet above, the line:

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

sets the context node to the `<ACT>` element. In the following line:

```
<xsl:value-of select="TITLE"/> has
```

the expression is relative to this context node. There are two ways of changing the context node in XSLT: `<xsl:template>` is one, and `<xsl:for-each>` is the other. These are the only ways of changing the context node. Using an XPath expression in a test attribute, for example, does not do this.

<xsl:sort>

This seems to be a good point to introduce `<xsl:sort>`. This element can be used inside `<xsl:apply-templates>` or `<xsl:for-each>` to alter the order in which nodes are processed. In most cases, when either of these statements selects these nodes, they will be processed in **document order** – that is, the order in which they were encountered when processing the document from top to bottom. This order can be altered using `<xsl:sort>`.

At the start of this chapter, we introduced a book catalog with just two books in it and a simplified structure. Now is a good time to use the full catalog, `catalog.xml`, which is available in the code download. Here are the first two books from this version:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--===== The Wrox Press Book Catalog Application =====>

<Catalog>
<Book id="1861003323">
  <Title>Professional VB6 XML</Title>
  <Authors>
    <Author>James Britt</Author>
    <Author>Teun Duynstee</Author>
  </Authors>
  <Publisher>Wrox Press, Ltd.</Publisher>
  <PubDate>March 2000</PubDate>
  <Abstract>Shows the VB community how to take advantage of XML
    technology and the available implementations</Abstract>
  <Pages>725</Pages>
  <ISBN>1-861003-32-3</ISBN>
  <RecSubjCategories>
    <Category>Programming</Category>
    <Category>Visual Basic</Category>
    <Category>XML</Category>
```

```

    </RecSubjCategories>
    <Price>$49.99</Price>
    <CoverImage>3323.gif</CoverImage>
  </Book>
  <Book id="1861003129">
    <Title>XSLT Programmer's Reference</Title>
    <Authors>
      <Author>Michael Kay</Author>
    </Authors>
    <Publisher>Wrox Press, Ltd.</Publisher>
    <PubDate>April 2000</PubDate>
    <Abstract>Learn how to use the XSLT language to develop
      web applications</Abstract>
    <Pages>800</Pages>
    <ISBN>1-861003-12-9</ISBN>
    <RecSubjCategories>
      <Category>Internet</Category>
      <Category>XML</Category>
      <Category>XSL</Category>
    </RecSubjCategories>
    <Price>$34.99</Price>
    <CoverImage>3129.gif</CoverImage>
  </Book>
  ...
</Catalog>

```

We will be using this quite a bit later in the chapter, but for now we will use it to see the effect of `<xsl:sort>`. Firstly, let's get a list of the book titles using `ShowTitles.xsl`:

```

<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

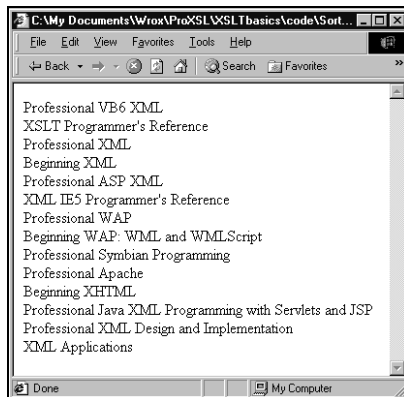
  <xsl:template match="/"><xsl:apply-templates/></xsl:template>

  <xsl:template match="Catalog">
    <xsl:for-each select="Book">
      <DIV><xsl:value-of select="Title"/></DIV>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>

```

When applied to `catalog.xml`, this gives the result:



If you look through the catalog, you will see that this is the order in which the books are listed. Now let's add an `<xsl:sort>` statement to the stylesheet (and re-save it as `CatalogSort.xml`):

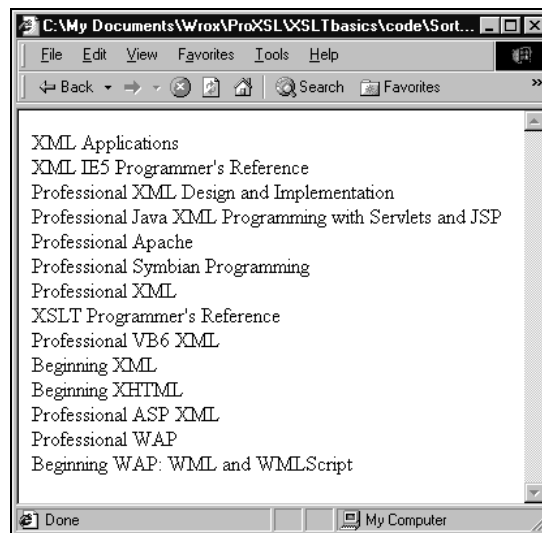
```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="Catalog">
    <xsl:for-each select="Book">
      <xsl:sort select="@id"/>
      <DIV><xsl:value-of select="Title"/></DIV>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The books are now listed in the order of their `id` attributes. These are their ISBN numbers without the hyphens. We could have just used the ISBN numbers, but the sort would then get confused if the style of hyphenation was not consistent. This is the result of using the stylesheet above on `catalog.xml`:



Inspection of the catalog will show that the titles are now listed in ISBN order.

There are various optional attributes of `<xsl:sort>` that control the sorting. We have already used `select`. Had we omitted this, the string value of the current node would have been used, rather than the `id` attribute. In this case, since the node has element content, this is not really a safe option (the string value would be the concatenation of the string values of the descendant nodes), but had we instead created a template for the `<Title>` element, we could have sorted on this by using the default value for the `select` attribute. The stylesheet might then have looked like this (`CatalogSort2.xml`):

```

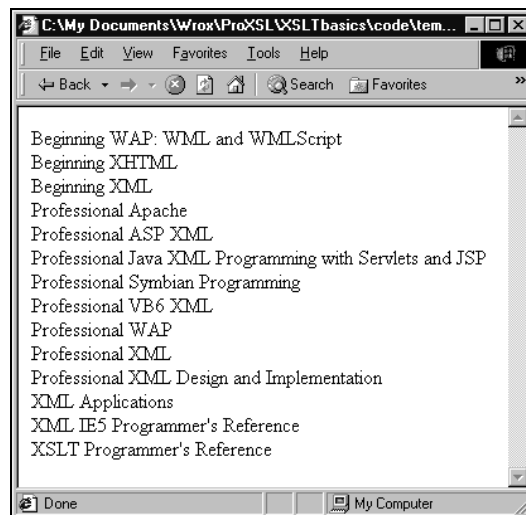
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:for-each select="Catalog/Book/Title">
      <xsl:sort/>
      <DIV><xsl:value-of select="."/></DIV>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>

```

And the result on transforming catalog.xml would have looked like this:



Because it is now applied to the <Title> elements, the line:

```
<xsl:sort/>
```

causes the titles to be output in alphabetical order.

Using the data-type attribute of <xsl:sort>, we can specify whether the data on which we are sorting is text (the default) or number. We sorted our ISBN values as text, which works in this case of fixed length strings, but in other cases, there would be a difference. The sequence of numbers from 1 to 20 sorted as text would be:

1,10,11,12,13,14,15,16,17,18,19,2,20,3,4,5,6,7,8,9

The order attribute has two possible values: ascending (the default) and descending, which are self-explanatory.

The `lang` attribute controls language-specific sorting orders. For example, in German, it is conventional to include the letter `ä` after the letter `a`, while in Swedish it would come after `z`. The possible values are the same as those of the `xml:lang` attribute and are specified in IETF RFC 1766 (<http://www.cis.ohio-state.edu/htbin/rfc/rfc1766.html>), "Tags for the Identification of Languages".

When sorting textual information, we can choose whether upper-case or lower-case values come first using the `case-order` attribute. This has values of `upper-first` and `lower-first`. If the former is used, when values start with the same letter, values starting with upper-case letters will occur in the output before any starting with lower-case letters. The default for this is language-dependent, and not specified in the recommendation. With no language specified, both XT and SAXON ignore case when sorting.

Sorting can be achieved on multiple criteria by using `<xsl:sort>` elements sequentially. In this case, sorting will occur first using the criteria defined by the first element, then sorted within these results using the later elements in order. If, for some strange reason, we wanted to sort all the speeches of Hamlet first in order of speaker, then by number of lines in the speech, then in alphabetical order of the text, we could use the following stylesheet, `HamletSort.xsl`:

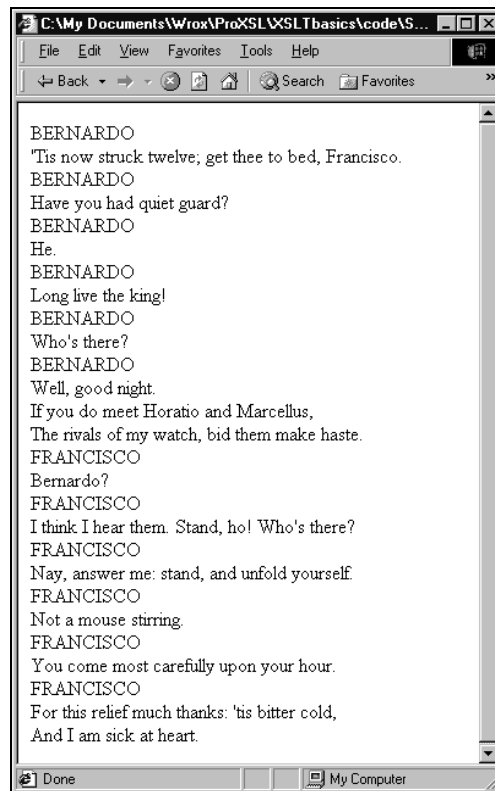
```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="SCENE"><xsl:for-each select="SPEECH">
    <xsl:sort select="SPEAKER"/>
    <xsl:sort select="count(LINE)"/>
    <xsl:sort select="LINE"/>
    <DIV><xsl:value-of select="SPEAKER"/></DIV>
    <xsl:for-each select="LINE">
      <DIV><xsl:value-of select="."/></DIV>
    </xsl:for-each>
  </xsl:for-each></xsl:template>

</xsl:stylesheet>
```

Using this just on our extract of the first scene (`HamletExtract.xml`), all Bernardo's speeches come before Francisco's, then for each speaker, all the one line speeches come before the two line speeches and within these criteria, the speeches are listed in alphabetical order. Notice that the apostrophe comes before any letters, as it does in English-language dictionaries:



<xsl:number>

The `<xsl:number>` element is used to determine the position of a node in the source tree. As such, it is useful for tasks such as numbering sections of a document and providing a table of contents. Various attributes allow control and formatting of the number produced.

The main control over the number produced is the `level` attribute. This can have one of three values, `single`, `any`, or `multiple`. The first of these is used to produce a number based on a node's position in relation to its siblings, for example, to number bullet points. The second is used to number nodes regardless of their level in the document hierarchy and can be used, for example, to number footnotes. And the last provides a number based on the level of a node in the hierarchy, such as you might use for numbering sections and sub-sections in a report.

Other attributes are available to say which nodes will be included in the numbering scheme and where in the source tree the numbering should start. As well as providing these controls over the number associated with a node, the `<xsl:number>` element provides formatting of the numbers. For example, when we use `value="multiple"`, we can specify that a section is to be numbered as 1.1.3 or A.1.3, or even 1.A.iii if we so desire. In fact, `<xsl:number>` provides all the flexibility in numbering and formatting of the number that we might expect from a word processor.

When we look at the `position()` function later in this chapter, we will look again at `<xsl:number>` to provide a comparison of the possible effects.

<xsl:text>

The `<xsl:text>` element is used to put a text string into the result tree. Isn't that what we have been doing by including the text in the stylesheet? Well yes, but there are two circumstances where we might need more control.

The first is when we are using entity references. If we include the text `<` in our stylesheet, what do we want written to the result tree? It could be the same text string, `<`; or it could be the single character `<`. The `disable-output-escaping` attribute lets us control this. If the value of this attribute is `no`, or the attribute is omitted, the result tree will contain the entity reference (`<`), but if the value is `yes`, the result will be the `<` character. This is useful if we are, for example, outputting a text string or code in some scripting language.

The second use of `<xsl:text>` is to control whitespace handling. Careful choice of examples has avoided this issue so far, but now is the time to see how XSLT controls the use of whitespace in its output. In general, a text node in the stylesheet is copied to the result tree, as we have seen in many of our examples. For example, we created a comma-separated list of book titles using:

```
<xsl:value-of select="Title"/>, <xsl:value-of select="PubDate"/>
```

Here we had a text node consisting of a comma followed by a space, and this was copied directly to the result tree.

Frequently, we will use whitespace in our stylesheets for formatting. For example, the code above could have been written as:

```
<xsl:value-of select="Title"/>,  
  <xsl:value-of select="PubDate"/>
```

In this case, the comma, any space characters that there might be after the comma, the newline character, and the several spaces used to indent the next line would all be output. However, since we were creating HTML and not embedding this within a `<PRE>` element, the HTML processor (in other words the browser) will convert these to a single space.

However, what about where we do not have a character such as a comma? The default behavior is that all text nodes that consist *only* of whitespace are ignored and not copied to the result tree. This allows us to use whitespace to format our stylesheet without worrying about it reaching our output.

There are two ways to vary this behavior. The first is in the XML source itself. If an ancestor element of the text node that contains only whitespace has an `xml:space` attribute with a value of `preserve` (and no intervening element has an `xml:space` attribute with a value of `default`), then whitespace will be preserved. This can give an equivalent result to the `<PRE>` element in HTML. If we want to control whitespace through the stylesheet itself, we use `<xsl:text>`. Within this element, all whitespace is preserved, even if a text node comprises only whitespace.

While on the subject of whitespace, if you are creating an HTML output you might be tempted to put in a non-breaking space character like this:

```
<xsl:value-of select="elem1"/>&nbsp;<xsl:value-of select="elem2"/>
```

Unfortunately, this is not well-formed XML since ` ` is not one of the built-in entity references in XML 1.0. (I put the version in here, in the hope that it will be included in a later version!)

Here are three ways around that problem. The first is to use a character reference instead:

```
<xsl:value-of select="elem1"/>&#160;<xsl:value-of select="elem2"/>
```

This is fine if you know you will remember what that reference is the next time you look at the stylesheet (and you are not expecting others to have to understand it easily).

The second is to use `<xsl:text>`:

```
<xsl:value-of select="elem1"/>
<xsl:text> </xsl:text>
<xsl:value-of select="elem2"/>
```

Again, this works, but is a bit long-winded. The third is to put a DTD internal subset into your stylesheet to define the entity:

```
<!DOCTYPE xsl:stylesheet [
  <!ENTITY nbsp "&#160;">
]>
```

Now you can use ` ` to your heart's content!

<xsl:element>

Up to now, when we have wanted to include an element in the result tree, we have just gone ahead and typed the start and end tags into the stylesheet. In most cases, that works. However, it will not work if the element name depends on data in the source XML document.

In this case we use `<xsl:element>` to create the element for us. This is used most frequently when creating an element-rich result tree from an attribute-rich source tree. However, we don't have an attribute-rich document to play with, so we will simply create element names based on the `id` attributes of the books in our catalog. We cannot use the `id` values themselves as element names. (Why not? Think back to your XML specification or wait to see at the end of this section.) So we will use the following code (`id.xsl`):

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates select="Catalog/Book"/>
  </xsl:template>

  <xsl:template match="Book">
    <xsl:element name="id{@id}"/>
    <xsl:text>
  </xsl:text>
  </xsl:template>

</xsl:stylesheet>
```


Notice the line:

```
<xsl:element name="id{@id}" />
```

The part in braces is an **attribute value template**, a feature of XSLT we will be meeting in the next chapter. This is the required format for the value of the name attribute.

Notice also the use of `<xsl:text>`:

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

The purpose of this was to put a line break between each element of the output. This is only there to make the resulting serialized document more readable. And here it is (when applied to `catalog.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<id1861003323/>
<id1861003129/>
<id1861003110/>
<id1861003412/>
<id1861004028/>
<id1861001576/>
<id1861004044/>
<id1861004583/>
<id186100303X/>
<id1861003021/>
<id1861003439/>
<id1861002858/>
<id1861002289/>
<id1861001525/>
```

There is another occasion you might want to use `<xsl:element>`. Because the element allows a namespace attribute, it provides more control over the namespace than we have when including the element as literal text in the stylesheet. Just as we have made the element name in the result tree dependent on the content of the input tree, we can do the same with a namespace. Like the name attribute, the value of this attribute is an attribute value template.

The final, optional, attribute of `<xsl:element>`, called `use-attribute-sets`, allows us to specify an attribute set to use with the defined element. Such sets of attributes can be defined using the `<xsl:attribute-set>` element.

Why couldn't we use the `id` values themselves as element names earlier? Our `id` values start with a digit – remember from the XML specification that an element name must start with an alphabetic character or an underscore or colon. Of course, we would never use a colon as this is used as a separator between a namespace prefix and local name.

<xsl:attribute>

You will probably find that you use <xsl:attribute> more often than <xsl:element>, particularly if you are creating HTML output.

This element allows us to create an attribute name and/or value in the result tree based on data in the source tree. For example, in our book catalog we have references to images of the book covers, such as:

```
<CoverImage>3323.gif</CoverImage>
```

If we were to display information from the catalog by creating an HTML page, we would want to create an element referring to this image. Something like:

```
<IMG src="covers/3323.gif">
```

We can do this using an attribute value template. The following stylesheet, covers.xsl, creates a table of book titles and images:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

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

  <xsl:template match="Catalog">
    <TABLE><xsl:apply-templates select="Book"/></TABLE>
  </xsl:template>

  <xsl:template match="Book">
    <TR>
      <TD><xsl:value-of select="Title"/></TD>
      <TD>
        <IMG>
          <xsl:attribute name="src">
            covers/<xsl:value-of select="CoverImage"/>
          </xsl:attribute>
        </IMG>
      </TD>
    </TR>
  </xsl:template>

</xsl:stylesheet>
```

Note that we have used the line:

```
<xsl:output method="html"/>
```

This ensures that the resulting `` element does not use the empty element syntax. The serialized output will have an element of the form:

```
<IMG src="covers/4583.gif">
```

rather than:

```
<IMG src="covers/4583.gif"/>
```

This ensures compatibility with a greater range of older browsers.

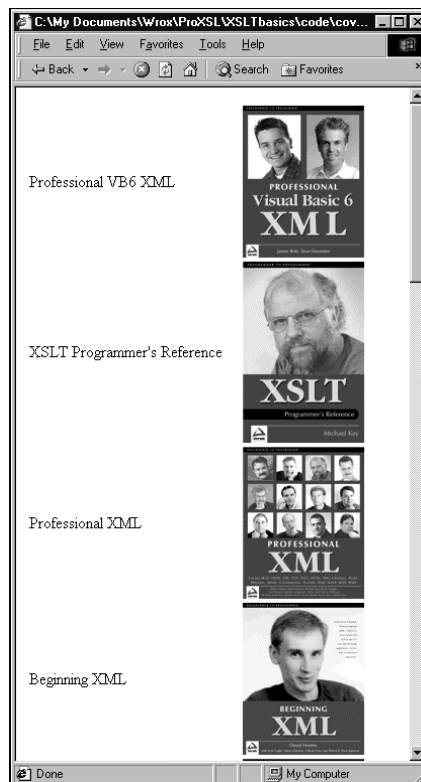
The part we are most interested in is:

```
<IMG>
  <xsl:attribute name="src">
    covers/<xsl:value-of select="CoverImage"/>
  </xsl:attribute>
</IMG>
```

Note that normally I would put the `<xsl:attribute>` element and all its content on a single line. This is because this element contains a text node (`covers/`), and so the additional whitespace will be included in the output. I have only reformatted this line to make it fit within the constraints of the printed page.

All the code above is doing is creating the `` element in the output with a `src` attribute using the value of the text node of the `<CoverImage>` element in the source tree. In this case, we have included some additional text to provide the correct relative URL for the images.

This screenshot shows the first four books:



Note that you would not use both an `<xsl:attribute>` and an `<xsl:element>` element. Instead you would use an attribute value template.

We have now covered the main elements of XSLT. With these, you can write many useful stylesheets. We will next look at two important aspects of XSLT – default templates and what happens when several templates match the same pattern.

Default Templates

Have you ever accidentally (or deliberately) applied an empty stylesheet to a document? For example, if we apply `empty.xsl`:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
```

to `shortcatalog.xml`, the result is:

```
<?xml version="1.0" encoding="utf-8"?>

  Designing Distributed Applications

    Stephen Mohr

  May 1999
  1-861002-27-0
  $49.99

  Professional ASP 3.0

    Alex Homer
    Brian Francis
    David Sussman

  October 1999
  1-861002-61-0
  $59.99
```

Although the stylesheet was empty, the output is not. The reason for this is that XSLT, as we saw briefly in Chapter 2, contains some default templates to match elements, attributes, and text nodes that are not matched elsewhere.

The main templates in this category are:

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

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

The first of these matches the document root and all elements. In both cases, it causes processing to continue by applying an `<xsl:apply-templates>` instruction to all child nodes. This means that, if you have a template matching an element buried deep in your source tree, but no template for one or more of its ancestors, the default template will cause processing to continue until that element is reached.

The second template matches text nodes and all attributes, causing the values of these to be output.

The combination of the first template, ensuring that the document root and all elements are processed, and the second, ensuring that all the text is copied to the result tree, is what generated our output.

There are also default templates for comments, processing instructions and namespace nodes, each saying "do nothing". Finally, in Chapter 4, we will meet another default template covering the `mode` attribute, which we have not met in detail yet.

Remember that these are only defaults. If you define your own templates, they will over-ride these. It could be that you are only processing a few elements in a document and don't want others processed. In this case, including the following line in your stylesheet will ensure that this happens:

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

Template Match Conflicts

When several templates match the same source node, the XSLT processor needs rules for choosing which template to use. In fact, because of the existence of the default templates, every stylesheet will contain multiple rules matching any node. Luckily, XSLT provides a clear set of rules for deciding which template to use.

We will see in Chapter 5 that one stylesheet can import another. If there is a conflict over which rule to apply when one belongs to the importing stylesheet and one to an imported stylesheet, the rule in the importing stylesheet is said to have higher **precedence**, and will be the rule applied. If the conflict is between rules in imported stylesheets, then stylesheets imported later have higher precedence than those imported earlier. Of course, imported stylesheets can import other stylesheets. If you get into this situation, the recommendation handles all cases using the concept of an **import tree**. Refer to the XSLT Recommendation itself to understand this.

The default rules are treated as though they were imported before the stylesheet itself, and so have the lowest possible import precedence.

Template rules can also have a **priority** assigned. We saw how to do this earlier, when discussing the `<xsl:template>` element. Where a priority is not assigned, a default priority value is used. These default priorities are in the range -0.5 to 0.5. In general, the more specific the match expression, the higher the priority.

If there is still a conflict after applying the precedence rules, the rule with the highest priority is applied.

It is possible that there will still be a conflict after the priority rules have been applied. This is treated as an error. The stylesheet processor will either indicate this error in some way or choose the last rule in the stylesheet that matches the pattern.

XSLT Functions

After that little aside to understand a couple of important areas of XSLT, we can get back to what XSLT can do for us. Now that we have covered the most important elements, we can cover the most important functions.

In Chapter 2, XPath core functions were introduced. In addition, XSLT has some functions of its own. While the core XPath functions are available to XSLT, the XSLT defined functions are not available to XPath when it is used beyond the confines of XSLT. The functions that we take a closer look at now are actually core XPath functions, but exist in the XSLT namespace.

The names of these functions usually give away what they do. Those we will cover here are:

- ❑ `position()`
- ❑ `last()`
- ❑ `name()`
- ❑ `count()`

We will also look at accessing nodes in a node-set by index.

position() and last()

In many cases, we will want to select nodes based on their position in a node-set. This is what we did in `ListCharacters.xsl` when we made the list of characters in Hamlet, to ensure that the last character did not have a comma after his name:

```
<xsl:if test="position() != last()">, </xsl:if>
```

As well as checking against `last()`, we can check against any index into the node-set. It is this that means we do not need a `first()` function to match the `last()` function. Instead, we just look for a `position()` of 1. To apply a template only if there is only one matching `elem` node, we would use:

```
<xsl:template match="elem[position()='1' and position()=last()]">
```

Note that the first matching node has an index of 1, and not 0.

As well as using the position within a test, we can use the value of the `position()` function to number our nodes. Earlier, we looked at `<xsl:number>`, which provides us with a number relating to a node's position in the source tree. In contrast, `position()` provides a number relating to the position in a node-set, which might be after a sort operation. It can therefore be used to provide a number relating to a node's position in the result tree.

We can very easily demonstrate this difference by using both methods to number the books in our catalog. We will produce a table of book titles with the results of numbering using `<xsl:number>` and `position()`.

This is our stylesheet, position.xsl:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40">

  <xsl:template match="/">
    <TABLE BORDER="1">
      <TR>
        <TD>Title</TD>
        <TD>position()</TD>
        <TD>xsl:number</TD>
      </TR>
      <xsl:apply-templates select="Catalog/Book"/>
    </TABLE>
  </xsl:template>

  <xsl:template match="Book">
    <TR>
      <TD><xsl:value-of select="Title"/></TD>
      <TD align="center"><xsl:value-of select="position()" /></TD>
      <TD align="center"><xsl:number/></TD>
    </TR>
  </xsl:template>

</xsl:stylesheet>
```

This is extremely simple. The template for the document root creates the table and applies the template for the <Book> elements. Each time the <Book> template executes, it adds a row to the table and puts in the book title with the result of applying <xsl:number> and position() to the current <Book> element.

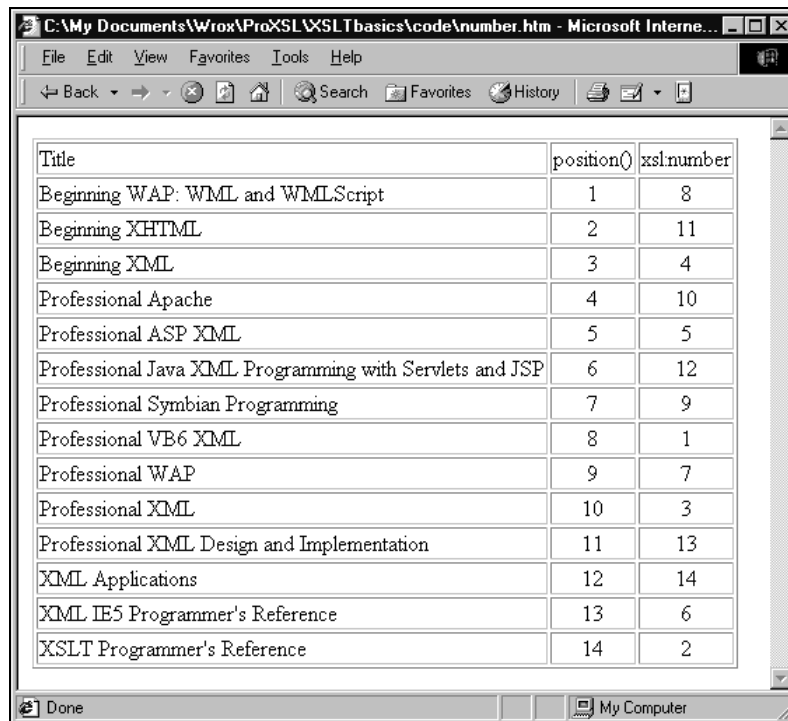
The result of applying this stylesheet to catalog.xml looks like this:

Title	position()	xsl:number
Professional VB6 XML	1	1
XSLT Programmer's Reference	2	2
Professional XML	3	3
Beginning XML	4	4
Professional ASP XML	5	5
XML IE5 Programmer's Reference	6	6
Professional WAP	7	7
Beginning WAP: WML and WMLScript	8	8
Professional Symbian Programming	9	9
Professional Apache	10	10
Beginning XHTML	11	11
Professional Java XML Programming with Servlets and JSP	12	12
Professional XML Design and Implementation	13	13
XML Applications	14	14

In this case, the results are the same since the node-set produced by the `select` attribute in the `<xsl:apply-templates>` element will be in document order. However, we can sort the catalog alphabetically by changing the template for the document root (`number.xsl`):

```
<xsl:template match="/">
  <TABLE BORDER="1">
    <TR>
      <TD>Title</TD>
      <TD>position()</TD>
      <TD>xsl:number</TD>
    </TR>
    <xsl:apply-templates select="Catalog/Book">
      <xsl:sort select="Title"/>
    </xsl:apply-templates>
  </TABLE>
</xsl:template>
```

Now the results will be somewhat different:



Title	position()	xsl:number
Beginning WAP: WML and WMLScript	1	8
Beginning XHTML	2	11
Beginning XML	3	4
Professional Apache	4	10
Professional ASP XML	5	5
Professional Java XML Programming with Servlets and JSP	6	12
Professional Symbian Programming	7	9
Professional VB6 XML	8	1
Professional WAP	9	7
Professional XML	10	3
Professional XML Design and Implementation	11	13
XML Applications	12	14
XML IE5 Programmer's Reference	13	6
XSLT Programmer's Reference	14	2

The `position()` function has produced numbering in the order of the node-set provided to the `<Book>` template (which is also the order in the result tree), while the `<xsl:number>` element has kept the numbering as it was in the source tree (so each title is numbered as it was in the previous example).

name()

The `name()` function merely returns the name of the node. If applied to a node-set rather than a node, the optional `node` parameter can identify the node required. Where there is more than one match, the function returns the first.

There are often occasions when you want to display a node name. For example, I often use XSLT stylesheets to document XML Schemas. In one of these, I want to display the facets of an element. Don't worry if you are not familiar with XML Schema, the important thing is that each facet is defined by its element name and the value of its `value` attribute. An extract of the stylesheet I use is:

```
<table>
  <xsl:for-each select="*">
    <tr>
      <td><xsl:value-of select="name()" />:</td>
      <td><xsl:value-of select="@value" /></td>
    </tr>
  </xsl:for-each>
</table>
```

We therefore list each facet name and value attribute value in two columns in a table.

count()

The final function we will consider here is the `count()` function, which returns the number of nodes in a node-set.

In the book catalog, `catalog.xml`, we could provide a template to list the titles of all the books with more than one author:

```
<xsl:template match="Book[count(//Authors)>1]">
  <xsl:value-of select="Title" />
</xsl:template>
```

Or, we might want to sort the books by the number of authors (`CountAuthors.xsl`):

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="Catalog">
    <xsl:for-each select="Book">
      <xsl:sort select="count(Authors/Author)" data-type="number" />
      <P>
        <xsl:value-of select="Title" /> has
        <xsl:value-of select="count(Authors/Author)" /> author(s)
      </P>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The result will be a list of books sorted in increasing number of authors order, and saying how many authors there are in each.

Summary

In this chapter, we have learnt the basics of XSLT through discussion and examples. In particular, we have seen:

- ❑ How XSLT is a declarative language with procedural elements, matching XPath expressions to apply templates to nodes in the source tree, while including control flow elements to provide the procedural aspects.
- ❑ The push and pull models, and where each is most suitable. We have also seen that most stylesheets will use some combination of these.
- ❑ A number of the elements used in XSLT, and how we can write useful stylesheets with just this subset of the language.
- ❑ A few of the functions built into the language, and how we can use these to give more control over our processing.
- ❑ How the built-in template rules ensure that all nodes are processed (unless we over ride them) and how XSLT knows which rule to apply when there is a conflict.

Before we move on to more advanced XSLT processing, why not try a few stylesheets of your own, perhaps gathering more statistics about Hamlet or displaying the book catalog in different ways?

