9

Transforming XML

Some talk about XML as a document format. Some talk about it as an underlying hierarchical model for storing data. At another level, an XML document can be perceived as data traveling through a network of processing agents. Each network node stores or processes the data and transmits the result to a neighbor node. In this world, an XML document is data flowing through or between applications on a network. However we look at it, any XML document is a collection of elements organized in accordance to a certain schema (whether explicit, through the use of a DTD or other schema, or implicit, without a defined schema) and is also a potential hierarchical structure. We can also say that the XML document is a serialized version of a hierarchical structure – a plain text document used for information exchange between processing agents. However, internally these processing agent do not use the serialized version (the XML document) but a more workable internal representation.

If XML is to truly help us create flexible applications that will talk to each other across platforms and in different applications, and if we really want to re-use the data we mark up in XML and share that data using XML as a common format, we need to be prepared for people and applications that do not use the same structure for their data we do. In this chapter we will look at ways in which we can transform the structure of our data into another XML vocabulary, or just re-order our own data.

In this chapter, we will concentrate on the transformation aspect of processing XML. There are many reasons why we need to transform XML, so we will start this chapter with a look at why and when we might want to transform XML into other forms. The majority of the chapter will use XSL transformations, although we will have a discussion at the end of the chapter about other methods for transformations.

The different transformation methods that can be used have caused heated debate in the XML community. Different programmers often prefer different solutions to how they transform XML, so, we will look at some of the different viewpoints underlying their preferences to let you decide which method to use (we do assume that you are either familiar with the DOM having read Chapter 5, or are already using it).

After looking at some reasons why you need to transform your documents, we will go on to look at using XSLT as a transformation language. We will introduce you to the basic syntax to get you used to using XSL for transforming documents. This will include an example of transforming our book list into a new structure. Note that this transformation requires knowledge of XPath, which is a specification used within XSLT to specify a particular part of an XML file. XPath was detailed in the previous chapter. Once we have looked at XSLT, we will look at how to use the DOM and script to alter the structure of the same book list. This will be followed by a look at more dynamic documents; taking what we have learnt from the XSL section of this chapter and the DOM chapter earlier, we will create a document that re-orders the contents of a table based on a user's interaction. We will then compare these two approaches to transformation. To wrap up the transformations we will look at when you might like to consider using the different approaches. In all, this chapter will cover:

- Why XML transformations are necessary
- An introduction to the XSLT syntax
- An example of using XSLT to transform static documents
- Using XSLT to transform more dynamic XML documents

This will put you in a position to choose the type of transformation that you need for your XML documents, and teach you the core concepts behind different types of transformation.

Why Transform XML?

If we are using XML stored in a text-based file, or we are receiving XML generated by some other type of program, it is in a fixed format. While XML is platform independent, and can be transferred between different parts of applications, there will be times when people require the information in different structures. In addition, there will be times when we need to transform a document's structure on the fly in an interactive document. For example, so that the document is restructured based upon a user's request or preferences. Transformations generally fall into one of three categories:

- **Structural transformations** – where you are transforming from one XML vocabulary into another, which is like a translation. An example may be between two financial markup languages such as FPML and finML.
- **Creating dynamic documents** – allowing users to re-order, filter, and sort parts of a document, such as allowing users to click on table column headings to reorder the contents of a table.
- **Transformations into a rendition language** – ready for presentation to a user in some form of browser, such as Wireless Application Protocol, HTML, VOXML, or Scalable Vector Graphics.

Let's take a look at each of these in turn.

Translating Between Different Vocabularies

If we think again about the catalog of books we marked up in XML in Chapter 2, there are many potential uses for the catalog data. For example, Wrox may use a catalog of its books on its web site and on its intranet using the DTD we created in Chapter 3. Meanwhile, several bookstores require almost exactly the same information. This all sounds like a perfect job for XML. However, if the different bookstores mark up their data using a different DTD to describe the same data, we need a way to transform our data into a version compatible with theirs.

For example, the www.wrox.com site may use the data marked up using the following section of the `pubCatalog.dtd` that we met in Chapter 3:

```
<Book ISBN="1-861003-11-0" level="Professional" pubdate="11-21-99"
      pageCount="500" authors="multi">
   <Title>Professional XML</Title>
   <Abstract>XML is an important area that you must learn about</Abstract>
   <RecSubjCategories>
      <Category>XML</Category>
      <Category>Programming</Category>
      <Category>Internet Programming</Category>
   </RecSubjCategories>
   <Price>$49.99</Price>
</Book>
```

However, XYZBooks Inc. may require the data in a different form, such as:

```
<Book>
   <Title>Professional XML</Title>
   <ISBN>1-861003-11-0</ISBN>
   <Abstract>XML is an important area that you must learn about</Abstract>
   <Pubdate>11-21-99</Pubdate>
   <RecSubjCategories>
      <Category>XML</Category>
      <Category>Programming</Category>
      <Category>Internet Programming</Category>
   </RecSubjCategories>
   <Price>$49.99</Price>
</Book>
```

As you can see, two of the attributes of the `<Book>` element have become elements in their own right: `<ISBN>` and `<Pubdate>`. Again, rather than preparing and storing the data for the two formats, we can just transform one into the other.

This is just one example of where we may need to transform one XML vocabulary into another, I am sure you can think of many others. One area where this sort of transformation has been touted as being important is in e-commerce, where different companies may need their data in different formats. Alternatively, we may even decide to update one of our existing applications, and need to be able to transform legacy XML (now there's a thought for you) into the new structure.

Transformations are a valuable player in XML, bearing in mind the intention that we should be able to **re-purpose** our data once it is marked up in XML. After all, if we only need to perform a simple transformation, there is no need to maintain two versions of our data. The transformation capabilities of XSL are ideally suited to this sort of transformation.

Dynamic Transformations

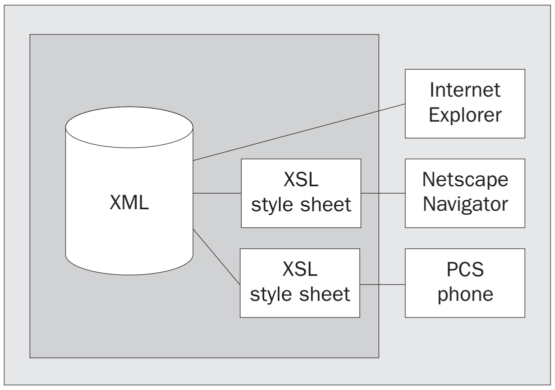
The last section looked at offering the same data in different ways, where both parties require specific, static versions of an XML document. But there are also occasions where we might need to do more dynamic transformations. If you think about spreadsheets, which undoubtedly revolutionized desktop PC use nearly twenty years ago, users require that data is re-sorted on the click of a column heading. This requires a more dynamic transformation.

Any transformation that requires user interaction, or producing interactive documents, can be quite a different task compared with producing static documents. Dynamic transformations often require event handling, which involves the use of a programming language.

Because scripting languages and the DOM allow transformations without XSL, and because the Document Object Model (DOM) can be used in browsers through its binding to JavaScript and other languages (such as Java, C++, Perl, Visual Basic, or Python) some people prefer to do these dynamic transformations with DOM and script (without XSL). Later in the chapter we will see examples of both and reasons why you may want to use one approach over another.

Different Browsers

Many web developers have experienced the headache of having to develop parallel sites, or parts of sites for incompatible browser versions. The idea that XML can only be served to web browsers that understand XML may seem like just another area for browser incompatibility to occur. However, if we were to develop our site content in XML, we could then transform it into different markup languages, so we could create different versions of HTML from the core XML content. Let's see how this might work:



Here we are using three different style sheets to create three versions of the XML content. The version for Internet Explorer 5 may still be in XML, while the other two can be two different rendition languages. This approach avoids the need for replicating the content three times for the different browsers. By transforming the XML data, several pages could use the same XML content to render pages in the correct format for the requesting browser. In this example we are simply using XSL style sheets as templates for how the data should be displayed, one for each client. These style sheets act as templates for the underlying data, therefore we could use the same style sheet to transform the data for several pages.

Indeed, transforming XML into HTML is popular where display in a browser is required, due to the lack of an XML linking specification from the W3C.

This approach is set to become increasingly important as new types of browsers make their way onto the Internet. We are already seeing digital television services, games consoles, and a variety of mobile devices, from handheld personal digital assistants (PDAs) to mobile phones, offering Internet access. As these diverse clients increase their share in the browser market there will be pressure to serve pages that are designed for their different needs. For example, as we will see in Chapter 14, the facilities offered by mobile phones with limited screen size and lower bandwidth and processing power will require special services. These may involve transforming the XML into another markup language such as the Wireless Markup Language (WML), which is designed for mobile phones and PDAs. So, the ability to transform our content into different versions will become increasingly common.

XSL

The **eXtensible Stylesheet Language** is an XML based language designed to transform an XML document into another XML document or to transform an XML document into rendition objects. The original XSL language has been split into three separate languages:

- Transformation (XSLT)
- Rendition (XSLF – which can involve the use of XSLT)
- Accessing the XML underlying structure (XPath)

XSL has its roots in both Cascading Style Sheets (CSS) and a language called DSSSL (Document Style Semantics and Specification Language (DSSSL – pronounced 'deessel'). As it evolves, XSL's styling aspects are becoming increasingly closer to CSS and farther from DSSSL. Styling is covered in Chapter 13.

As you might have guessed, the key area we are looking at in this chapter is the transformation capabilities of XSL. The XSLT specification became a Recommendation on 16th November 1999. It also relies on the XPath specification, which became a recommendation on the same day, as a way of selecting the area of the document to transform. We looked at XPath in the last chapter starting on page 326.

XSLT

This section looks at how we can use XSLT to transform XML documents, and we will see where XPath is employed in XSLT. As the first sentence of the XSLT specifications explicitly states: "*[XSLT] is a language for transforming XML documents into other XML documents*". As we saw earlier, we may need to transform XML into another structure for many reasons. To do this we need an XSLT processor. We will discuss two of the commonly available ones as soon as we have seen what the XSLT processor actually does.

XSLT is a language written in XML. This means that an XSLT style sheet for transforming an XML document is actually a well-formed XML document. So, in this chapter we will be learning the syntax of XSLT and what it allows us to do.

First we have a very important point to clarify:

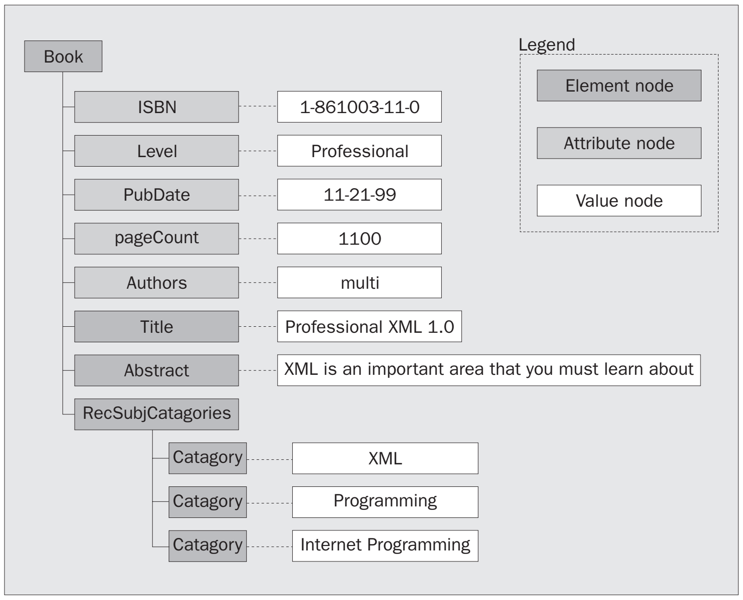
XSLT engines do not manipulate documents; they manipulate structure.

In order for an XSLT engine to be able to transform an XML document, the document must first be converted into a **structure** or an **internal model**. This internal model is a tree. This model is independent of any API used to access it. In the SGML world, this abstract model is called a **grove**. Because, actually, XML is a subset of SGML, it also inherits some of its basic concepts. Thus, the grove is simply the abstract tree structure independently of any API used to reach or manipulate the tree's entities. For instance, the DOM is the API recommended by W3C to access the grove. The DOM is the API, the grove is the abstract structure. Thus, a grove may have more than one API or could have different APIs for different languages. Throughout this chapter, we refer to the grove when we're talking about the abstract tree structure.

So the following XML:

```
<?xml version="1.0" ?>
<Book ISBN="1-861003-11-0" level="Professional" pubdate="11-21-99"
      pageCount="500" authors="multi">
   <Title>Professional XML</Title>
   <Abstract>XML is an important area that you must learn about</Abstract>
   <RecSubjCategories>
      <Category>XML</Category>
      <Category>Programming</Category>
      <Category>Internet Programming</Category>
   </RecSubjCategories>
   <Price>$49.99</Price>
</Book>
```

Can be represented in an abstract form of a tree something like this:



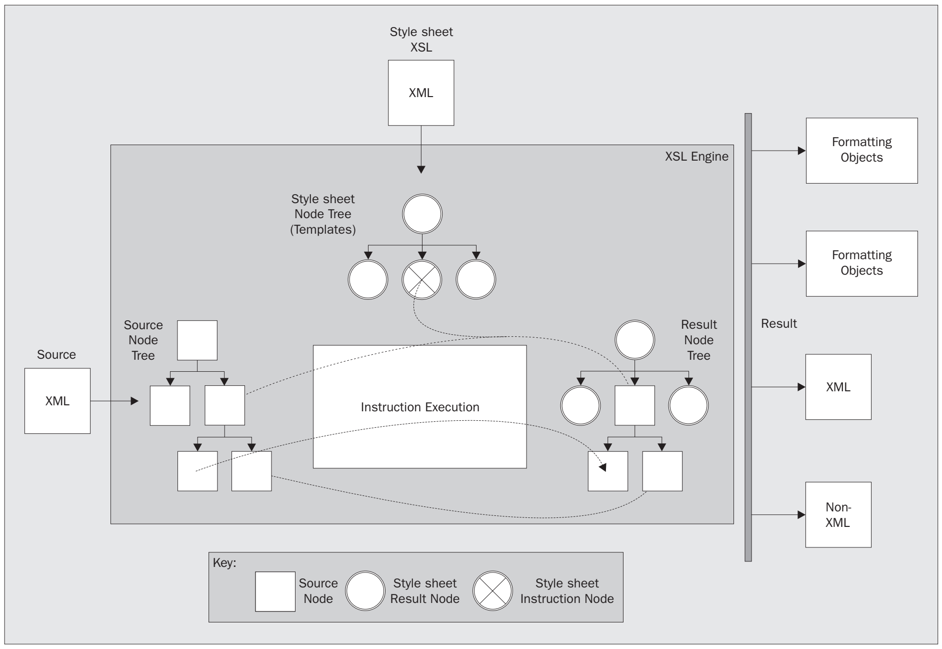
It doesn't matter how we choose to look at or process the file, `<Title>`, `<Abstract>`, `<RecSubjCategories>`, and `<Price>` are all children of `<Book>`, and `<Category>` is a grandchild of `<Book>` and a child of `<RecSubjCategories>`. This is why we say this abstract tree structure is a model that is independent of any API used to access it (such as the W3C DOM), and it is this **structure** that XSL processors use, to select the appropriate part of the structure. XPath is the language used to access any element of the tree structure.

How Does the XSL Processor Transform the Source Document?

As we said, XSLT operates on the document model not the syntax. Both the source and destination formats are applications of XML, and the underlying structure of both is a tree. In addition the XSL style sheet is an XML document, thus it too can be represented by a tree. So, in all, XSLT processors hold three trees.

XSLT is a **declarative** language, which means that you specify how you want the result to look, rather than saying how it should be transformed, and this is why we need the XSL processor to do the work. The XSL style sheet is made up of **templates** that specify how each node of the source tree should appear in the result tree.

The following diagram illustrates how the processor works:



Here we can see that there are the three structures. Remember that the source and the result structures are abstract representations of the document. The processor goes through the source grove, starting with the root, and looks for a matching template in the style sheet tree. When it finds one, it uses the rules in the template to write an abstract representation of the result into the result tree. Then it moves through the source document, node by node, lead by the XSLT instruction `<xsl:apply-template>`, looking for a match in the style sheet. If there is no matching template, it moves on to the next one. We can say that it executes a default template, which has no output result. Then, the result tree is translated into an XML document, text, an HTML document, or whatever the desired result is.

This, at least in theory, is what should happen. But there are different variations on how XSLT engines are built. XSLT engines could be optimized, and the style sheet may not necessarily be stored as a grove or tree structure. However, this gives a general idea of their behavior.

Having taken an overview of what an XSL processor does in order to perform its transformation, you will need to make sure you have one installed on your machine to work with the examples in this chapter. As there are different ways of implementing an XSL processor, let's take a look at two of them:

- **MSXML** – the Microsoft XML parser with a DOM interface, which includes an XSLT engine in the form of a COM component. The MSXML engine included in IE5 is quite out of date compared to the recommendations. The technology preview version is more up to date.
- **XT** – James Clark's dedicated XSLT engine. This is written in Java and is therefore usable on several platforms. XT is more up to date concerning its conformance with the latest XSLT specifications.

The MSXML XSL Processor

MSXML is not just a parser; it also includes an XSL processor. MSXML is tightly integrated with Internet Explorer 5, but is also available as a standalone COM component from http://msdn.microsoft.com/xml/ for integration with applications. This component uses the DOM to manipulate the abstract tree structure of the XML document. Because of this, it can be interfaced to languages like JavaScript, Delphi, Visual Basic, VisualCOBOL, VBScript, PerlScript, PythonScript, C++, etc. This component requires that at least Internet Explorer version 4 or above is present on the system, since the Microsoft XSLT engine has some dependencies on other DLLs provided by this application.

The original DOM interface is defined using a CORBA Interface Definition Language (IDL), but Microsoft's component technology COM uses a different IDL, so the MSXML component interface is defined using the COM IDL. The Microsoft DOM implementation, nevertheless, respects the spirit of this recommendation by keeping the same method names in the object's interfaces. The interface `IXMLDocument` is equivalent to the W3C DOM level 1 interface named `document`. The W3C `document` interface inherits from the `node` interface, and similarly the `IXMLDocument` interface inherits from the `IXMLNode` interface. The `IXMLDocument` interface has also been extended to contain additional methods that aid XML document parsing and transformation.

For example, the following ASP script uses the `IXMLDocument` interface of the MSXML component to parse both the XML document to be transformed and the XSL style sheet. Then, it transforms the parsed source XML document using the parsed XSL transformation sheet:

```
<%@ LANGUAGE = VBScript %>
<%
   ' Set the source and style sheet locations
   sourceFile = Server.MapPath("catalog.xml")
   styleFile = Server.MapPath("catalog.xsl")

   ' Load the XML and get it parsed
   Set source = CreateObject("Microsoft.XMLDOM")
   source.async = false
   source.load(sourceFile)
```

```
    ' Load the XSLT and get it parsed
    Set style = CreateObject("Microsoft.XMLDOM")
    style.async = false
    style.load(styleFile)

    ' Transform the XML document using the XSL transformation sheet.
    Response.Write(source.transformNode(style))
%>
```

The general mechanism used to transform documents with MSXML is:

- Load the original document to be transformed. The `load()` method also parses the document so that the document is stored internally as a tree structure (like the one we saw earlier).
- Load the XSLT document. Again, the `load()` method parses the document and transforms it into a tree.
- Make the transformation by using the `transformNode()` function. This function returns a string (a `BSTR`). The returned string contains the transformed document. So, if the XSLT transformation sheet contains an XML to HTML transformation, then the document stored in the resultant string is an HTML document.

The MSXML component integrates in the same component:

- An XML parser
- An extended DOM level 1 interface to the tree
- An XSLT transformation engine

The XT XSL Processor

XT is another popular XSLT processor; written by James Clark, it is simple to use and can be downloaded for free from the author's site http://www.jclark.com/xml/xt.html. It is written in Java, and has been successfully tested on several Java Virtual machines. For the Win32 platform, it can be downloaded as a single executable, although this requires Microsoft's Java VM to be installed on the machine. This processor will be useful to experiment with the different transformations introduced throughout this chapter.

As opposed to MSXML, which includes its own XML parser, the XT engine can operate with any SAX-compliant parser (we discussed SAX in Chapter 6). As long as the SAX parser is implemented in Java, it will interface properly with the XT engine. The package also comes with a fast Java parser named XP.

- XT is used through the command line. On Windows, running XT is probably easier than on other platforms. The following command line transforms an XML document with an XSLT style sheet, resulting in an HTML document:

```
Xt booklist.xml booklist.xsl booklist.htm
```

XT also accepts XSLT parameters, for example:

```
Xt booklist.xml booklist.xsl booklist.htm result=HTML
```

In the example above, the `result` parameter is included in the XSLT style sheet as an XSLT variable. This variable can then be used in the XSLT templates.

A big advantage of XT is that it can run on platforms other than windows. However, not all platforms allow you to run a Java application as a stand-alone executable, and most platforms require that you run the Java application with the Java executable provided with the Java JDK. For instance, the following command line will run XT on Linux:

```
java -Dcom.jclark.xsl.sax.parser=xp.jar com.jclark.xsl.sax.Driver booklist.xml
booklist.xsl booklist.htm result=HTML
```

The XT engine speed depends a lot on the processing power of your machine, since Java is an interpreted language.

Using XSLT

To see how XSLT works, we are going to jump straight in with a simple example. Let's start with the details of a book marked up in XML and transform it into XHTML for display in a browser.

XHTML is an extended version of HTML 4.0 designed to be an application of XML. For more information, consult the latest W3C recommendation proposal at http://www.w3.org/TR/xhtml1.

Here is some book catalog information marked up in XML, according to the DTD developed in Chapter 3:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--======= The Wrox Press Book Catalog Application =======-->
<Catalog>
<Book>
   <Title>Designing Distributed Applications</Title>
   <Authors>
      <Author>Stephen Mohr</Author>
   </Authors>
   <Publisher>Wrox Press, Ltd.</Publisher>
   <PubDate>May 1999</PubDate>
   <Abstract>5 principles that will make your web applications more flexible
            and live longer</Abstract>
   <Pages>460</Pages>
   <ISBN>1-861002-27-0</ISBN>
   <RecSubjCategories>
      <Category>Internet</Category>
      <Category>Programming</Category>
      <Category>XML</Category>
   </RecSubjCategories>
</Book>
```

```
<Book>
   <Title>Professional Java XML</Title>
   <Authors>
      <Author>Alexander Nakhimovsky</Author>
      <Author>Tom Myers</Author>
   </Authors>
   <Publisher>Wrox Press, Ltd.</Publisher>
   <PubDate>August 1999</PubDate>
   <Abstract>Learn to utilize the powerful combination of Java and
             XML</Abstract>
   <Pages>600</Pages>
   <ISBN>1-861002-85-8</ISBN>
   <RecSubjCategories>
      <Category>Java</Category>
      <Category>Programming</Category>
      <Category>XML</Category>
   </RecSubjCategories>
</Book>
</Catalog>
```

Let's have a look at the simple XSLT style sheet that will be used to transform the source document into the required result document, which will be an XHTML document that displays the titles of the books in the catalog:

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

<xsl:output method="html"/>

<xsl:template match="/">
   <html>
      <head>
         <title>The book catalog</title>
      </head>
      <body>
         <xsl:apply-templates select="//Book" />
      </body>
   </html>
</xsl:template>

<xsl:template match="Book">
   <DIV style="margin-left: 40pt;
        margin-bottom: 15pt;
        text-align: left;
        line-height: 12pt;
        text-indent: 0pt;" >
      <xsl:apply-templates select="Title" />
   </DIV>
</xsl:template>

<xsl:template match="Title">
   <DIV style="margin-left: 40pt;
        font-family: Arial;
        font-weight: 700;
        font-size: 14pt;" >
      <SPAN>
```

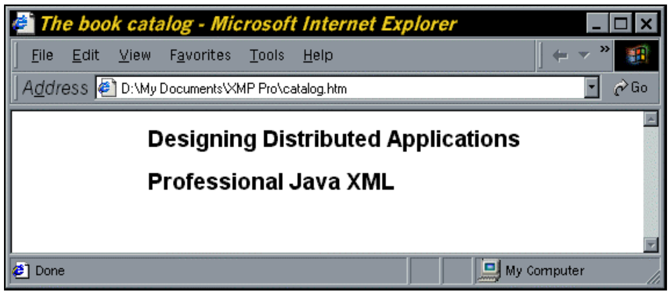
```
            <xsl:value-of select="."/>
         </SPAN>
      </DIV>
   </xsl:template>

   </xsl:stylesheet>
```

Caution: as it stands, this example can only be executed by an XSLT engine conformant to the XSLT version 1 recommendations. SAXON and XT fall into this category, IE5.0 doesn't – because it does not support XPath and some XSLT constructs. However, in this particular case it is possible to modify the above file such that IE5.0 can use it: you will have to change the namespace from http://www.w3.org/1999/XSL/Transform *to* http://www.w3.org/TR/WD-xsl *and remove the* `<xsl:output method="html"/>` *statement. Bear in mind, though, that this modification will not work for all the examples in this chapter, so until there a more up to date parser is supplied for IE you're better off using XT.*

Finally, the resulting XHTML document will look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
   <title>The book catalog</title>
</head>
<body>
   <DIV style="margin-left: 40pt;
               margin-bottom: 15pt;
               text-align: left;
               line-height: 12pt;
               text-indent: 0pt;">
      <DIV style="margin-left: 40pt;
                  font-family: Arial;
                  font-weight: 700;
                  font-size: 14pt;">
         <SPAN>Designing Distributed Applications</SPAN>
      </DIV>
   </DIV>
   <DIV style="margin-left: 40pt;
               margin-bottom: 15pt;
               text-align: left;
               line-height: 12pt;
               text-indent: 0pt;">
      <DIV style="margin-left: 40pt;
                  font-family: Arial;
                  font-weight: 700;
                  font-size: 14pt;">
         <SPAN>Professional Java XML</SPAN>
      </DIV>
   </DIV>
</body>
</html>
```

The simple result looks like this:


The book catalog - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address D:\My Documents\XMP Pro\catalog.htm
Go
Done
My Computer

Designing Distributed Applications

Professional Java XML

Getting Help with the Transformation

To better understand what's supposed to be going on in this example, pretend that you are an XSLT engine for a moment, and look at the world through its eyes (go on, no one is watching). First, as an XSLT engine, remember that you need the document structure not the text itself. After all, you can only process the structure not the text. So, somebody has to transform the text into the required abstract tree structure, the grove. As an XSLT engine, you may have one of these two friends:

- ❑ A parser with a DOM interface
- ❑ A parser just giving you an event for each element

If you use the services of a parser having a DOM interface, this implies that the parser encapsulates the tree and that you access any of the objects on the grove through the DOM interface.

If you interface with a parser giving you an event for each element, you manage the grove yourself and then store the document's structure the way you want. This is the approach in Java using a SAX interface.

Therefore, my dear XSLT engine, you have the choice of whether to get help from your parser with a DOM interface and outsource the grove management to them, or to manage it yourself.

The internal structure could be implemented in different ways; however, even if you use associative arrays or lists pointing to lists, the modeled structure is a tree. The DOM is the W3C recommendation that specifies how to interface to this structure.

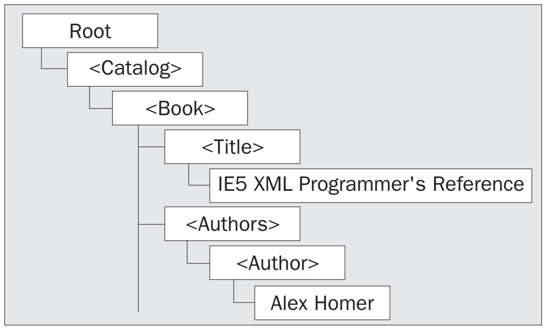
If you remember back to the earlier diagram, you are holding three trees. One contains the representation of the source document, one will hold the representation of the result tree's structure, but what was the third tree for? Wasn't an XSLT document also an XML document? Yes. You're beginning to learn your role well. This is what the third tree is for. In the case of the XSLT document, its conversion to an internal tree structure may not imply a conversion from text into a hierarchical structure. The XSLT internal structure may be something else, something more optimized for XSLT processing.

So, the original XML document is first parsed then converted into an abstract tree structure, an internal representation of the hierarchical structure. The DOM is an interface to this internal structure. The XSLT document is also parsed and converted into an internal structure. It may be an abstract tree structure but may be also another kind of structure, optimized for template processing and pattern matching.

The `Catalog.xml` file:

```
<Catalog>
<Book>
   <Title>IE5 XML Programmer's Reference</Title>
   <Authors>
      <Author>Alex Homer</Author>
   </Authors>
   <Publisher>Wrox Press, Ltd.</Publisher>
   <PubDate>August 1999</PubDate>
   <Abstract>Reference of XML capabilities in IE5</Abstract>
   <Pages>480</Pages>
   <ISBN>1-861001-57-6</ISBN>
   <RecSubjCategories>
      <Category>Internet</Category>
      <Category>Web Publishing</Category>
      <Category>XML</Category>
   </RecSubjCategories>
</Book>
</Catalog>
```

will be represented by an abstract tree in the XSL processor as shown below:



How the Style Sheet Transforms the Document

As we said, XSL is an application of XML, so the style sheet (or transformation sheet, if you prefer) is really an XML document. Because it is an XML document, it can start with the XML declaration, which indicates to a XML parser in which XML version this document is encoded.

The root element of our style sheet is the `<xsl:stylesheet>` element:

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

The first attribute of the `<xsl:stylesheet>` element is the XSLT version. The second is the attribute `xmlns:xsl`, which holds the namespace for the XSL transformation recommendation.

As you might remember from Chapter 7, on Namespaces and Schemas, this declares the namespace of XSLT. As you can see, the prefix associated with the namespace is `xsl`, so the root element is actually `<stylesheet>`, but it has been qualified by `xsl:`, its namespace prefix. Having declared the namespace, any element beginning with the prefix `xsl:` is part of the XSL vocabulary.

The `<stylesheet>` element contains three **templates**, each of which is nested within the `<template>` element, which is actually `<xsl:template>` in the style sheet because we included the namespace. You will notice that the `<template>` element has an attribute called `match`. The value of this attribute is a pattern that matches the node of the tree that the template should be applied to, in the form of an XPath expression.

The first task is to tell the XSLT engine the desired output. In the example, an HTML result is expected, and specified using:

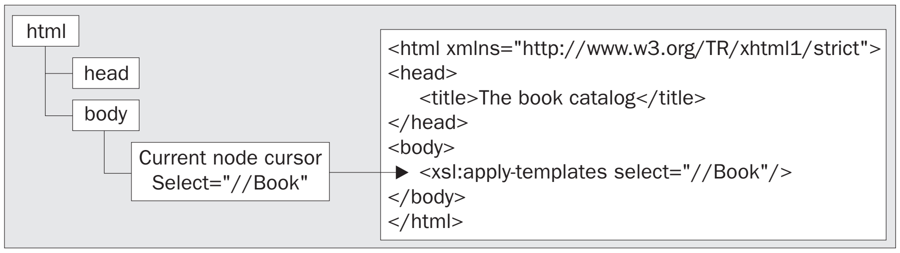
```
<xsl:output method="html"/>
```

Knowing the expected output format, as a processor, you will start in the source document tree at the root node. You will then look for a template in the style sheet that matches the root node. Note that the root node is the document node, not the first element. In the example case, the root node is not the `<Catalog>` element, but rather the XML document itself. So, do we have a template matching the document root? The answer is yes. If you remember back to the XPath section in the last chapter, the root of a document can also be represented by a forward slash (/) symbol. This is exactly what we see in the first template:

```
<xsl:template match="/">
   <html xmlns="http://www.w3.org/TR/xhtml1/strict">
      <head>
         <title>The book catalog</title>
      </head>
      <body>
         <xsl:apply-templates select="//Book" />
      </body>
   </html>
</xsl:template>
```

So, you have found a template that matches the root element of the source document. What do you do now? To better represent what's happening in the head of an XSLT processor, imagine a cursor navigating in the original XML document nodes tree; its position is the **current node**, and right now the current node is the root element.

Step 1: You have positioned the current cursor on the root node and found a matching template in the XSLT structure. The template has a "/" pattern. So, output the following result. Remember, we are working with an abstract tree structure, represented in the diagrammatic part of the following figure, on the left:



```
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head>
    <title>The book catalog</title>
</head>
<body>
 ▶ <xsl:apply-templates select="//Book"/>
</body>
</html>
```

In the middle of this first template, nested in the `<Body>` element, there is an `<xml:apply-templates />` construct. This is where we will be writing the content of the page. It has an attribute called `select`, whose value is an XPath expression. This construct means:

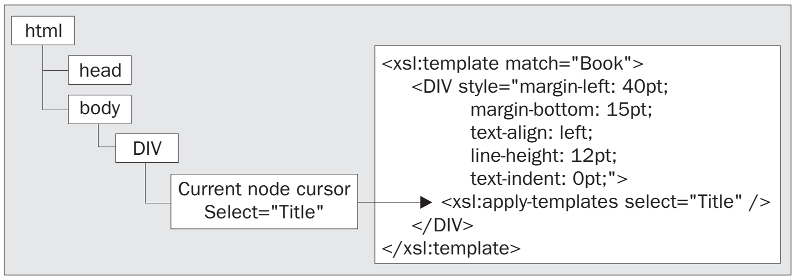
- "From the XPath query "`//Book`", obtain a node list. Then, for each node in this node list, try to match a template. If a match is found, apply the template."

But what does "`//Book`" mean? It means "select the `<Book>` elements that are descendants of the root node".

Step 2: Further down in our XSL file we find a template match for `<Book>` elements (`<xsl:template match="Book">`), so next we apply this template to the first `<Book>` element in our node list. Our current cursor is now on the first `<Book>` node.

Step 3: We then insert the `<Book>` matching template's content where the `<apply-templates select="//Book" />` construct is located. First we add the default CSS styling properties for the book data – any further elements that do not contain more specific CSS style properties will inherit these properties. Next, we find another `<apply-templates />`, this time with a `select` attribute of "`Title`". From the rule in Step 1, we know that this requires us to construct a node set of `<Title>` nodes. However, this time the current node is the first `<Book>` node, and our XPath expression dictates that our new node set will only contain the `<Title>` nodes that are children of the current node. This means that our node set will consist of the `<Title>` child of the first `<Book>` node.

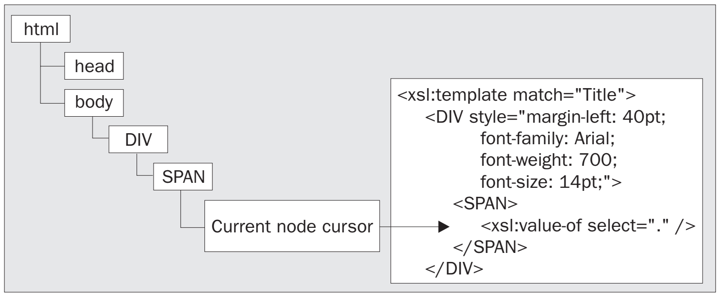
This is where we start to see the versatility of the XPath expressions we use. If we were to substitute `<xsl:apply-templates select="//Title" />` for the existing element the node set would contain all `<Title>` descendants of the root node (the parent of the `<Book>` node) – which would mean all the `<Title>` nodes in the grove.



```
<xsl:template match="Book">
    <DIV style="margin-left: 40pt;
            margin-bottom: 15pt;
            text-align: left;
            line-height: 12pt;
            text-indent: 0pt;">
        <xsl:apply-templates select="Title" />
    </DIV>
</xsl:template>
```

Step 4: Next we try to match a template against the `<Title>` node in our node set. Again we find a match: the `<xsl:template match="Title">` template. We then insert the contents of this template where the `<xsl:apply-templates select="Title" />` construct is located.

Step 5: The contents of the `<Title>` matching template consist of some more CSS styling properties and an `<xsl:value-of select="."/>` element. This construct pulls the values of the nodes specified by its `select` attribute XPath out of the tree structure. In this case the XPath is "`.`", meaning the `<Title>` node itself, so we write out the contents of the `<Title>` node.



Step 6: We have now applied templates to all the nodes in the node set we created in the `<Book>` matching template, so we move on to the next node in the first node set we created – which is the second `<Book>` node. This is treated in the same way as our first `<Book>` node, so we go through steps 4 and 5 again. We continue this process until we have processed all of the `<Book>` elements.

During this process we not only transformed an XML document form a certain document type to another, but also performed some editing in the process – only the `<Book>` and `<Title>` elements are transformed. Also, the transformation is not a one-to-one transformation. For each element of the original document there can be more than one element in the resultant document.

We should note something important here. The template matching the `<Title>` element does not insert elements into the result tree, it inserts text nodes. As we said earlier, a grove is an internal hierarchical structure. When the XML document is converted into this hierarchical structure, we transform the text document into a tree like model. In this tree, child elements are also the tree's child nodes. Data content is a child node too. For instance, the `<Book>` element contains one `<Title>` element. This `<Title>` element does not contain elements but contains data content, which becomes a child node.

So, what have we learned while pretending to be an XSL engine?

❑ First we build a grove that is an internal tree representation of the document. This grove always has a root element. The root element represents the XML document – it is not the document's top level element. Then, under that root element is the node hierarchy. Each node is typed. A node can be, for example, a node for the DTD, for the schema, or for a processing instruction. If an element has attributes, then each element also has a collection of attribute nodes. If the element has data content, then a data content node is added under the element node. Hence, an element node may have a collection of attribute nodes and a data content node as children.

❑ Secondly, we create a structure for the XSLT document. This may also be a grove but it may be any kind of structure optimized for template processing and pattern matching.

❑ Then, each time we encounter an `<xsl:apply-templates>` element we form a node list and continue processing with this list. If the `<xsl:apply-templates>` element contains a `select` attribute, we obtain the node list from the XPath query specified, Otherwise the node list will consist of all child nodes.

❑ Each time an `<xsl:value-of>` construct is encountered, we extract a value from the source tree based on the XPath expression in the `select` attribute.

❑ Transformation is not solely restricted to one-to-one translation, it also allows addition of new information content, one-to-many element translation, element addition, and element deletion.

XSLT Style Sheet Structure

So, we have seen our first example of an XSLT document transforming an XML document into a new structure. Now we shall take a general look at how XSLT documents are structured.

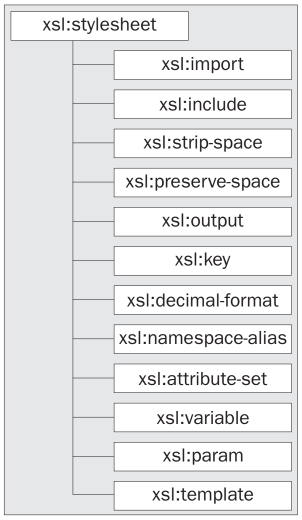
We learnt a lot from our first example; not only have we seen how the XSLT processor works through a document that it must transform – which will be very useful as we look at creating other elements – we have also used four of the key XSLT elements:

- `<xsl:stylesheet>`
- `<xsl:template>`
- `<xsl:apply-templates>`
- `<xsl:value-of>`

We started our first example document with the XML declaration because the style sheet is an XML document. Remembering that `xsl:` is used as a qualified name prefix for all elements that are part of the XSLT namespace, the `<stylesheet>` element is the document element that contains the other elements of the style sheet, and this is where the namespace is declared. Within this element we had three `<template>` elements, which are used to specify how the element, or other node, specified in the `match` attribute should be transformed. These can be seen as the main building blocks of most transformations. The `<xsl:apply-templates>` element is used to tell the processor to process all child elements of the current element if no `select` attribute is present. Otherwise, only the element nodes matching the selection criteria are processed. Finally, the `xsl:value-of` element is used to write out element content.

This illustrates the two types of element defined in the XSLT specification. Apart from the root element, there are **templates** and **instructions**. `<xsl:template>` is, obviously, a template, as it would appear underneath the root element in the abstract tree structure, while `<xsl:apply-templates>` and <`xsl:value-of>` are instructions that appear as children of the `<template>` element. Remember that an XSLT document is an XML document, and as such can be converted into a tree structure.

The following diagram shows the top-level elements, children of the `<xsl:stylesheet>` element:



This illustrates how the `<xsl:stylesheet>` element is always the root element of any XSLT style sheet. Beneath this element we can have any of these top-level elements. So, the abstract structure that the XSLT processor works upon would be like this for the root and any top-level elements. Let's look through some of these.

Creating Templates

The key construct in any XSLT style sheet is the `<template>` element, which is used with a `match` attribute, whose value is a pattern – or XPath expression – saying which node the template should be applied to. More specifically, any XPath expression returning a node list is a candidate for a `match` attribute value. However, it's easier to remember that the pattern is an XPath expression indicating the nodes on which templates should be applied. Within the templates we can include elements, and element's content.

Dealing with White Space

To help us deal with white space, XSL provides two constructs, which are used as top-level elements:

- `xsl:strip-space` to remove selected nodes that are just white space
- `xsl:preserve-space` maintains any white space in the content

xsl:strip-space

The `<xsl:strip-space>` element removes text nodes that just consist of white space from the tree when the element name is included in the `elements` attribute. For instance, the following `<xsl:strip-space>` element will remove any `<BOOKLIST>` or `<ITEM>` element's text node that consist only of white space:

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

Thus, the above element tells the XSL engine that if the elements `<BOOKLIST>` and `<ITEM>` are made up of white space, their text node should be removed from the tree (though the element node remains).

xsl:preserve-space

Similarly, if we want instead to preserve some white space for certain element content, then we include the `<xsl:preserve-space>` element. Again, the `elements` attribute is used to specify the list of all elements that we want to add to the set of elements with their white space preserved. In the example below white space is preserved for the `<CATALOG>` and `<PRICE>` elements.

```
<xsl:preserve-space elements="CATALOG PRICE"/>
```

Output Format

The `<xsl:output />` element can be used to specify the output format of a result tree (although it is not required that an XSL processor implement this function).

Again, this is a top-level element and should normally immediately follow the `<xsl:stylesheet>` element. This is not a mandatory element, and the XSL engine will have, in many cases, the default set to HTML if some conditions are met:

- The root node of the result tree should have a child
- The root's first child node should be an `html` element
- Any nodes preceding the first child node should contain only white space characters.

It is possible also to set the result tree to different formats like `xml`, `html`, or `text`. An interesting attribute of the `<xsl:output>` element is the `encoding` attribute. This latter allows us to translate from a certain encoding to another one if the target encoding is supported by the XLST engine. For instance, an ASCI encoded XML document can be transformed into a Unicode encoded document. Thus, to transform a XML document into a new XML document with a different encoding, you should include the `<xsl:output>` element immediately after the `<xsl:stylesheet>` element:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" encoding="UTF-16"/>
```

Combining Style Sheets

One convenient way to re-use code is to create modules. These modules can then be re-used in other modules – XSLT can include or import external style sheets. Two constructs are used to this effect:

- The `<xsl:include>` element
- The `<xsl:import>` element

xsl:include

The `<xsl:include>` element simply allows us to include an external style sheet where the `<xsl:include>` element is located. The XSLT document referred by the URI is first parsed then the children of the included document's `<stylesheet>` element replace the `<xsl:include>` element in the including document. It is necessary that the `<xsl:include>` construct be located as child of the `<stylesheet>` element:

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

xsl:import

`<xsl:import>` is quite different from `<xsl:include>` – `<xsl:include>` just means perform a file inclusion, while `<xsl:import>` modifies the document's tree. In fact, the `<xsl:import>` construct modifies the templates' order and processing precedence.

First and foremost, the element should precede any other top-level element – it should be the first child of the `<xsl:stylesheet>` element.

At first, all style sheets that are imported are included as text. Once they have all been collected, they are used to form an **import tree**. Thus, each style sheet imported is included in the host style sheet import tree. It is possible to have imported style sheets that themselves import other style sheets.

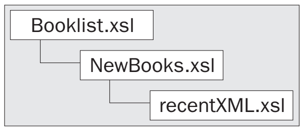
For example, the `booklist.xsl` style sheet might import a second style sheet like so:

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

Now let's say that the `newBooks.xsl` imports another style sheet:

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

Then the resultant **import tree** would look like this:



This forms a structure in which directives from one style sheet can take precedence over another, where instructions in `Booklist.xsl` take priority over both others. When templates are matched to a particular element, `Booklist.xsl` is processed first, then `NewBooks.xsl,` and finally `recent.xsl`. The `<xsl:import>` construct has direct impact on the style sheet processing. The XSL document tree is modified by this element, and the style sheets are assembled into a single unit – the import tree.

Embedding Style Sheets

A style sheet is not necessarily a separate document. It can also be embedded into another XML document. For instance, a dynamically constructed XML document may include its style sheet before transmitting the document to the user agent. In the example below, an XSL style sheet is embedded in a XML document:

```
<?xml version="1.0"?>
<?xml-stylesheet href="#BooklistStyle" type="text/xsl" media="screen"?>
<xsl:stylesheet version="1.0"
                id= "BooklistStyle"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   ...
</stylesheet>
<BOOKLIST>
   ...
</BOOKLIST>
```

Here, the style sheet is included in the XML document. It is referenced using an XML `id` (indicated by the # symbol), so the XSL processor knows that the style sheet is within a document fragment uniquely identified by an `id` attribute on an element. The XSL processor will then extract the style sheet fragment from the document, parse it, and construct the internal structure necessary for XSL processing. The XML document itself is parsed, but the resultant document tree does not contain the style sheet tree. So, from a single XML document the user agent makes two document structures:

- A grove for the XML document excluding the `<xsl:stylesheet>` element (which encloses the whole style sheet). This structure is accessible using the DOM.
- A structure for the XSL document, which includes only the `<xsl:stylesheet>` element and its contents. The structure may or may not be a grove and may or may not be accessible using the DOM.

Examples of Using XSLT

As you have already seen, XSLT is a powerful tool for transforming an XML document with a certain structure or document type in to a new one – such as transforming an XML document into XHTML. Having seen an early example that taught us how the XSLT processor works, and having seen a reference section explaining the most commonly used elements available to us in XSLT, the second half of this chapter will look at some more examples of using XSLT in different situations. These will include:

- Structural transformation, from one XML vocabulary to another
- Repeated processing of elements using a loop – `xsl:for-each`
- Sorting order of elements to be processed
- Conditional processing using `xsl:if` and `xsl:choose`
- Creating dynamic documents

Structural Transformations

Let's have a look at an example that allows us to change the structure of the XML file into another XML structure, rather than XHTML. Say we need to re-order the elements of an XML document brought to you by a colleague, as shown below:

```
<?xml version="1.0"?>
<BOOKLIST>
   <ITEM>
      <CODE>16-048</CODE>
      <CATEGORY>Scripting</CATEGORY>
      <RELEASE_DATE>1998-04-21</RELEASE_DATE>
      <TITLE>Instant JavaScript</TITLE>
      <PRICE>$49.34</PRICE>
   </ITEM>
   <ITEM>
      <CODE>16-105</CODE>
      <CATEGORY>ASP</CATEGORY>
      <RELEASE_DATE>1998-05-10</RELEASE_DATE>
      <TITLE>Instant Active Server Pages</TITLE>
      <PRICE>$23.45</PRICE>
```

```
    </ITEM>
    <ITEM>
       <CODE>16-041</CODE>
       <CATEGORY>HTML</CATEGORY>
       <RELEASE_DATE>1998-03-07</RELEASE_DATE>
       <TITLE>Instant HTML</TITLE>
       <PRICE>$34.23</PRICE>
    </ITEM>
 </BOOKLIST>
```

So far so good, but he also added some spice to your life by adding a number of requirements:

- The document must be published on a browser able to render XML documents with CSS style sheets
- Each item (which is a book) must be displayed as a block
- Each title should be displayed first (in that block)
- The category and the code should be displayed on the same line but with the category displayed first
- The last line of each block should contain first the release date and then the price

And, as if that was not enough, he mentions that the `<CATEGORY>` content should be indicated with a "Category:" string, and that the code should be enclosed in parentheses. Sounds like he's added enough spice to your life to make a hot Mexican meal? Well, the release date and the price should also be separated with a "-". And, the cherry on top of the sundae, you can use only CSS1 style sheets. At this point you may think that this is not your day. But XSLT is just waiting to help you.

So, the first thing that you need to do, so that you can style the document using CSS, is to transform the existing document structure into something that looks like this:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/css"
   href="catalog.css"
   media="screen"?>
<BOOKLIST>
   <ITEM>
      <TITLE>Instant JavaScript</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: Scripting</CATEGORY>
         <CODE>(16-048)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>Release date: 1998-04-21</RELEASE_DATE>
         <PRICE>Price: $49.34</PRICE>
      </LISTING>
   </ITEM>
   <ITEM>
      <TITLE>Instant Active Server Pages</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: ASP</CATEGORY>
         <CODE>(16-105)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>release date: 1998-05-10</RELEASE_DATE>
         <PRICE>Price: $23.45</PRICE>
      </LISTING>
   </ITEM>
   <ITEM>
      <TITLE>Instant HTML</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: HTML</CATEGORY>
         <CODE>(16-041)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>release date: 1998-03-07</RELEASE_DATE>
         <PRICE>Price: $34.23</PRICE>
      </LISTING>
   </ITEM>
</BOOKLIST>
```

To do this, we will be using the following style sheet, which we will study in more detail in a moment. It only contains two templates.

Note: The example can be processed with XT or SAXON. To run the following template on Microsoft Internet Explorer, you need a more recent version of the MSXML component than the one provided with Internet Explorer version 5.

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

<xsl:output method="xml"/>

<xsl:template match="/">
   <xsl:processing-instruction name="xml-stylesheet">
      href="catalog.css" type"text/css" media="screen"
   </xsl:processing-instruction>
   <BOOKLIST>
      <xsl:apply-templates/>
   </BOOKLIST>
</xsl:template>

<xsl:template match="ITEM">
   <ITEM>
      <TITLE>
         <xsl:apply-templates select="TITLE/text()" />
      </TITLE>
      <DESCRIPTION>
         <CATEGORY>
            Category:
            <xsl:apply-templates select="CATEGORY/text()" />
         </CATEGORY>
         <CODE>
            (<xsl:apply-templates select="CODE/text()" />)
         </CODE>
         </DESCRIPTION>
         <LISTING>
            <RELEASE_DATE >
               Release date:
               <xsl:apply-templates select="RELEASE_DATE/text()" />
            </RELEASE_DATE>
            <PRICE>
               - Price:
               <xsl:apply-templates select="PRICE/text()"/>
            </PRICE>
         </LISTING>
      </ITEM>
   </xsl:template>

   </xsl:stylesheet>
```

As we saw in the previous example, the source document has first to be transformed into a grove (an internal hierarchical structure). The elements are then matched (or not as the case may be) to templates after the XSLT document has been converted into an internal structure for processing.

The first template is matched to the document's root:

```
   <xsl:template match="/">
      <xsl:processing-instruction name="xml-stylesheet">
         href="catalog.css" type"text/css" media="screen"
      </xsl:processing-instruction>
      <BOOKLIST>
         <xsl:apply-templates/>
      </BOOKLIST>
   </xsl:template>
```

XML documents can be associated with style sheet documents using an `<?xml-stylesheet ... ?>` processing instruction. We want our resultant document to be associated with a CSS style sheet, so we have to write it into the template so that the resulting document includes the processing instruction.

To create the processing instruction in the result tree, we use a special XSL construct, the `<xsl:processing-intruction>` element. The attribute name provides the processing instruction name and the data content all the other attributes. Thus the following XSL element:

```
      <xsl:processing-instruction name="xml-stylesheet">
         href="catalog.css" type"text/css" media="screen"
      </xsl:processing-instruction>
```

is converted in the result tree to:

```
   <?xml-stylesheet href="catalog.css" type"text/css" media="screen"?>
```

The other `<BOOKLIST>` elements included in this template will be inserted in the result tree. The now familiar `<apply-templates>` construct indicates to the XSLT processor that it should process all children without any selection criteria – the children to be processed are the current node children – then these children are matched with templates, or if a child consists of data content that is not matched to a template, it is inserted in the result tree. Otherwise, if the data content child is matched to a template, then this template is processed and its content included in the result tree.

As you may have noticed, there is no template matching the <BOOKLIST> element. XSLT engines have an implicit template matched to any element without an explicitly specified template. This implicit template allows successful recursive processing to continue in the absence of a successful pattern match with an explicitly specified template. The implicit template is defined as:

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

Thus the <BOOKLIST> element, which does not have an explicitly defined template rule, is matched to the implicit template. This implicit template is also called the **default template**.

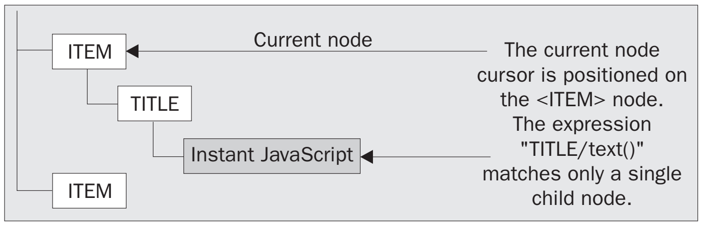
The <BOOKLIST> element contains <ITEM> elements, for which there is a template. In fact, this is the element that we want to reorganize. Reorganizing the <ITEM> elements is quite easy, we just include the elements sorted the way we want. If new elements have to be added, we simply include them in the template as well.

We use the <xsl:apply-templates> construct quite differently from how we used it in the first template. Earlier we used the select attribute to specify to the XSLT engine that only elements matching the selection criteria will be matched to an explicit template or to the default template.

The following expression includes the <TITLE> data content of the original XML document into the produced <TITLE> element in the output tree.

```
      <TITLE><xsl:apply-templates select="TITLE/text()"/></TITLE>
```

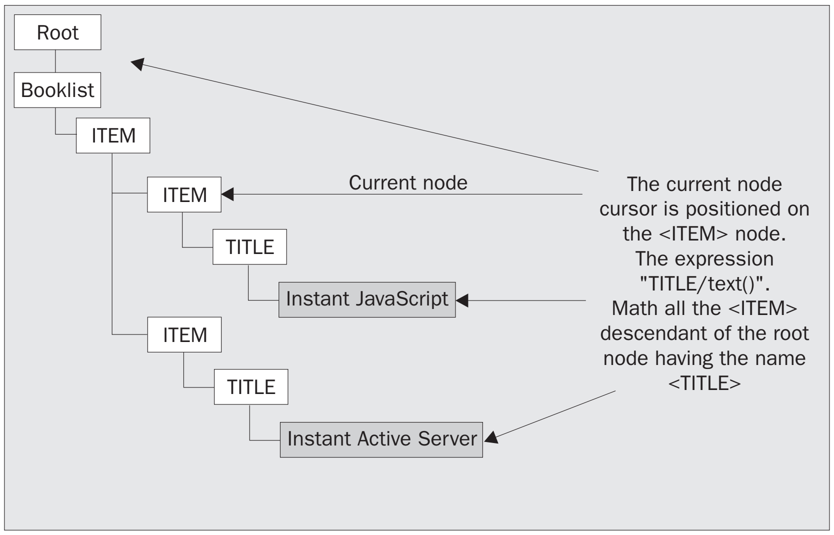
The select attribute XPath expression indicates to the XSLT engine that the content of the text child node of the <TITLE> element will be inserted at the same position as the <xsl:apply-templates> construct is located.


ITEM
Current node
The current node cursor is positioned on the <ITEM> node. The expression "TITLE/text()" matches only a single child node.
TITLE
Instant JavaScript
ITEM

Note here that the template is matched to the <ITEM> element, and because the <TITLE> element is a child of this, the right XPath expression is "TITLE/text()". If we used "//TITLE/text()" instead then the data content of all the <TITLE> elements would have been inserted in the result tree as shown below:

```
<TITLE>Instant JavaScriptInstant Active Server PagesInstant HTML</TITLE>
```

This is because adding the "//TITLE" means "process all descendant nodes of the root node (of type "element") that are named <TITLE>". Note that the XPath expression ".//TITLE/text()" means "process all descendents of the currently selected node having the name <TITLE>". The . in front of the // makes all the difference.


Root
Booklist
ITEM
ITEM
Current node
TITLE
Instant JavaScript
ITEM
TITLE
Instant Active Server
The current node cursor is positioned on the <ITEM> node. The expression "TITLE/text()". Math all the <ITEM> descendant of the root node having the name <TITLE>

Thus, all the `<xsl:apply-template select...>` constructs contained in the template matching the `<ITEM>` node are relative to the currently selected node. In our case, this is the `<ITEM>` node. The current node cursor was moved to the `<ITEM>` element by the template's match attribute:

```
<xsl:template match="ITEM">
   <ITEM>
      <TITLE>
         <xsl:apply-templates select="TITLE/text()" />
      </TITLE>
      <DESCRIPTION>
         <CATEGORY>
            Category:
            <xsl:apply-templates select="CATEGORY/text()" />
         </CATEGORY>
         <CODE>
            (<xsl:apply-templates select="CODE/text()" />)
         </CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE >
            Release date:
            <xsl:apply-templates select="RELEASE_DATE/text()" />
         </RELEASE_DATE>
         <PRICE>
            - Price:
            <xsl:apply-templates select="PRICE/text()"/>
         </PRICE>
      </LISTING>
   </ITEM>
</xsl:template>
```

As we saw earlier, there is also an alternative way to extract the right information from the original XML document. This is illustrated in the following example, where all `<xsl:apply-templates.../>` constructs are replaced by `<xsl:value of .../>` constructs:

```
<xsl:template match="ITEM">
   <ITEM>
      <TITLE>
         <xsl:value-of select=".//TITLE"/>
      </TITLE>
         <DESCRIPTION>
         <CATEGORY>
            Category:
            <xsl:value-of select=".//CATEGORY"/>
         </CATEGORY>
         <CODE>
            (<xsl:value-of select=".//CODE"/>)
         </CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE >
            Release date:
            <xsl:value-of select=".//RELEASE_DATE"/>
         </RELEASE_DATE>
         <PRICE>
            - Price:
            <xsl:value-of select=".//PRICE"/>
         </PRICE>
      </LISTING>
   </ITEM>
</xsl:template>
```

We just saw that there are two ways to insert the right elements in the right place:

❑ With the `<xsl:apply-templates>` construct

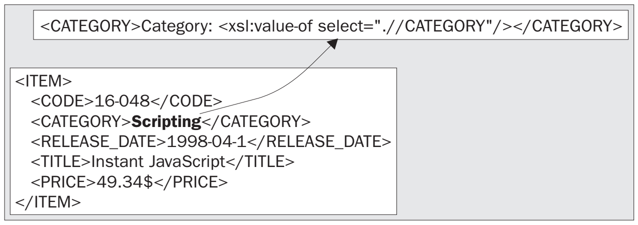
❑ With the `<xsl:value-of>` construct

I recommend the second construct – `<xsl:value-of>` – which explicitly tells us that it is the value of the selection that is included in the output tree. As you noticed, we do not have to include the "`text()`" instruction in the selection expression since the value of an element is its data content.

New data content can also be added to the pulled content. For instance, we want to include the expression Category: at the beginning of the resultant data content so that we obtain something like:

```
         <CATEGORY>Category: Scripting</CATEGORY>
```

Again, we use the `<xsl:value-of ... />` construct, again it is replaced by the `<CATEGORY>` element's data content, but we have also added the text Category:.


<CATEGORY>Category: <xsl:value-of select=".//CATEGORY"/></CATEGORY>
<ITEM>
<CODE>16-048</CODE>
<CATEGORY>**Scripting**</CATEGORY>
<RELEASE_DATE>1998-04-1</RELEASE_DATE>
<TITLE>Instant JavaScript</TITLE>
<PRICE>49.34$</PRICE>
</ITEM>

Using XSLT to transform documents is very important for a number of areas. Here we have used it to transform one XML document into another structure so that we can display it in the appropriate way. While we were essentially using the same tags in the result document, we were re-inserting their content from the template using the `<value-of>` or the `<apply-templates>` construct to obtain the source document element data, and writing in the elements ourselves. We could just as easily be creating new tags so that the document would be transformed into a completely new vocabulary. We could also transform this document into the vocabulary used in the earlier example (which we developed in Chapter 2).

This kind of technique would therefore be ideal for transforming to presentation languages such as HTML and WML (Wireless Markup Language). It is also useful for translating to different vocabularies if people we exchange information with require a different XML structure. For example, if we were exchanging financial data, and one company used FPML while the other used FinXML, we could translate between the two.

Repetition

Loops are the kind of construct we often use in procedural programming languages. XSLT also supports a loop construct, in the form of the `<xsl:for-each ... />` element. Its content is repeated as long as there are elements in the original XML document that correspond to the value of the `select` attribute. As an example, we can use the `for-each` construct to transform the booklist XML document into an XHTML document where items are listed in a table. This is actually the whole XSLT style sheet – you will notice some interesting things about its use:

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

<html xmlns="http://www.w3.org/TR/xhtml1/strict"
      xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <head>
      <title>The book catalog listed in a table</title>
   </head>
   <body>
      <table border="1" cellspacing="0" cellpadding="5">
         <tbody>
            <xsl:for-each select="BOOKLIST/ITEM">
               <tr>
                  <th align="left"><xsl:value-of select=".//TITLE"/></th>
                  <td><xsl:value-of select=".//CATEGORY" /></td>
                  <td><xsl:value-of select=".//RELEASE_DATE" /></td>
                  <td><xsl:value-of select=".//PRICE" /></td>
               </tr>
            </xsl:for-each>
         </tbody>
      </table>
   </body>
</html>
```

The first thing you'll probably notice about this example is that it uses a format different from that in previous examples. This document only contains a single template, which *implicitly* matches the root element. Actually, there is no need in this alternative format to include the `<xsl:template>` construct. The implicit template is, in this case, `<xsl:template match="/">`.

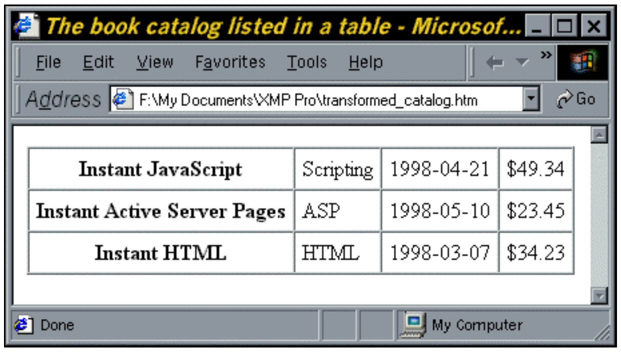
The table that we are creating contains a row for each `<ITEM>` (or book). In this example we do not use the template matching mechanism, we loop through the document using the `for-each` construct, extracting element content with `value-of` constructs:

```
<xsl:for-each select="BOOKLIST/ITEM">
   <tr>
      <th align="left"><xsl:value-of select=".//TITLE"/></th>
      <td><xsl:value-of select=".//CATEGORY" /></td>
      <td><xsl:value-of select=".//RELEASE_DATE" /></td>
      <td><xsl:value-of select=".//PRICE" /></td>
   </tr>
</xsl:for-each>
```

We are telling the processor: "for each `<ITEM>` element nested in the root `<BOOKLIST>` element, write the contents of the `<TITLE>`, `<CATEGORY>`, `<RELEASE_DATE>`, and `<PRICE>` elements out into the table". The selection criterion for the loop construct is an XPath expression starting from the root; so we have to explicitly include all elements included in the document tree's branch up to the `<ITEM>` element. The loop finishes when no more elements satisfy the selection criterion.

To obtain the value included in a table cell, we use the `<xsl:value-of ... />` element. As you may remember, this has the effect of extracting the data content from the node matching the XPath expression that is the value of the `select` attribute.

The resulting output is shown below:


The book catalog listed in a table - Microsof...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\transformed_catalog.htm
Go
Done
My Computer

Instant JavaScript	Scripting	1998-04-21	$49.34
Instant Active Server Pages	ASP	1998-05-10	$23.45
Instant HTML	HTML	1998-03-07	$34.23

Sorting

After having moved elements in our booklist document, even having added new data content and linked the resultant document to a CSS style sheet, let's add sorting. The goal is to sort the booklist items first by category, then by title.

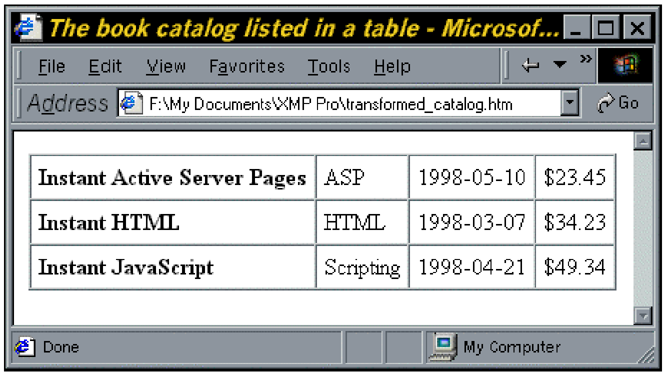
The XSL construct used for this task is the `<xsl:sort>` element. To tell the XSLT engine which element to sort, we include the `select` attribute set to an XPath value. For instance, to sort the items by title we would use the following construct:

```
<xsl:sort select=".//TITLE"/>
```

The question is: where can we include this element? It is an instruction element that can only be used inside the `<xsl:apply-templates ... />` or `<xsl:for-each ... >` elements, as illustrated in the following style sheet, which is a modification of the one in the *Repetition* section:

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

<html xmlns="http://www.w3.org/TR/xhtml1/strict"
      xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <head>
      <title>The book catalog listed in a table</title>
   </head>
   <body>
      <table border="1" cellspacing="0" cellpadding="5">
         <tbody>
            <xsl:for-each select="BOOKLIST/ITEM">
               <xsl:sort select=".//CATEGORY"/>
               <xsl:sort select=".//TITLE"/>
               <tr>
                  <th align="left"><xsl:value-of select=".//TITLE"/></th>
                  <td><xsl:value-of select=".//CATEGORY" /></td>
                  <td><xsl:value-of select=".//RELEASE_DATE" /></td>
                  <td><xsl:value-of select=".//PRICE" /></td>
               </tr>
            </xsl:for-each>
         </tbody>
      </table>
   </body>
</html>
```

Now, the `<xsl:for-each ... >` element contains new instructions on how to process selected nodes. The engine will sort the nodes before matching them with templates. In the above example, the nodes are sorted first by category, then by title – as shown below:

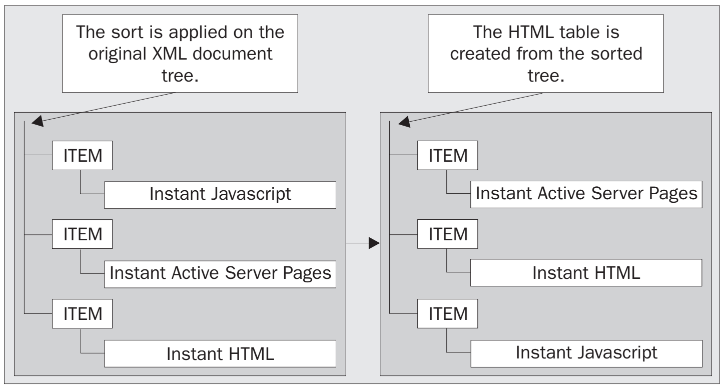

The book catalog listed in a table - Microsof...
File
Edit
View
Favorites
Tools
Help
Address
F:\My Documents\XMP Pro\transformed_catalog.htm
Go
Done
My Computer

Instant Active Server Pages	ASP	1998-05-10	$23.45
Instant HTML	HTML	1998-03-07	$34.23
Instant JavaScript	Scripting	1998-04-21	$49.34

The sort order is dependent on the `<xsl:sort .../>` element order. For instance, the following construct will instead sort the nodes by `<TITLE>` and `<RELEAE_DATE>`:

```
               <xsl:sort select=".//TITLE"/>
               <xsl:sort select=".//RELEASE_DATE"/>
```

It is important to remember that the `<sort>` construct re-orders the nodes before any other processing is imposed on the node.


The sort is applied on the original XML document tree.
The HTML table is created from the sorted tree.
ITEM
Instant Javascript
ITEM
Instant Active Server Pages
ITEM
Instant HTML
ITEM
Instant Active Server Pages
ITEM
Instant HTML
ITEM
Instant Javascript

Conditional Processing

Other constructs often found in procedural languages are:

- The `if` construct, named `<xsl:if>` in XSLT
- The `if/elseif` construct, named `<xsl:choose>` in XSLT

At this point you may think that for a declarative language XSLT includes several procedural constructs. This is true; what makes it a declarative language is that you do not have to explicitly tell the XSLT engine to output a certain content. You specify a content or template to it, which will be included in the output result. Nonetheless, XSLT also has certain procedural characteristics.

Now, let's say that in our previous example we only wanted to include the `<ITEMS>` part of the `Scripting` category in the result tree. To do so, we need a filter, or an `if` construct, to indicate to the engine "if you encounter *this* pattern then do *this*". To achieve this result, we include the `<xsl:if>` instruction element in our template, as in the example below:

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

<html xmlns="http://www.w3.org/TR/xhtml1/strict"
      xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <head>
      <title>The book catalog listed in a table</title>
   </head>
   <body>
      <table border="1" cellspacing="0" cellpadding="5">
         <tbody>
            <xsl:for-each select="BOOKLIST/ITEM">
               <xsl:if test="contains(CATEGORY/text(), 'Scripting')">
                  <tr>
                     <th align="left"><xsl:value-of select=".//TITLE"/></th>
                     <td><xsl:value-of select=".//CATEGORY" /></td>
                     <td><xsl:value-of select=".//RELEASE_DATE" /></td>
                     <td><xsl:value-of select=".//PRICE" /></td>
                  </tr>
```

```
                    </xsl:if>
                 </xsl:for-each>
              </tbody>
           </table>
        </body>
     </html>
```

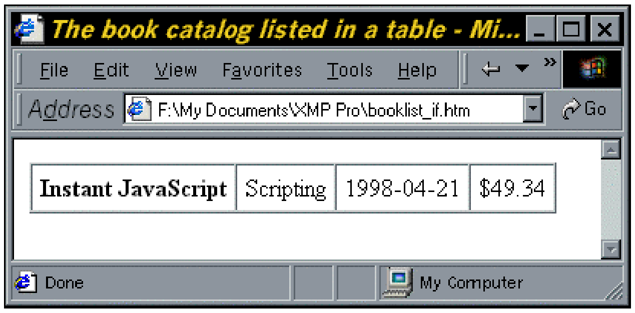
Now, in the loop we included a condition to be satisfied. If the condition is true, then the template included in the `<xsl:if>` element is inserted in the result tree. If not, then the template is simply ignored.

In the `test` attribute, we compare the string `Scripting` to the `<CATEGORY>` element's data content. In fact, we use the `contains()` function to check if the `<CATEGORY>` element's text node contains the `Scripting` string. First, the `contains(string1, string2)` function returns a Boolean value of true if the first string (`string1`) contains the second string (`string2`), as we saw in the *Predicate* section in the last chapter. `string1` is obtained by the `CATEGORY/text()` XPath expression, which uses the address of the `<CATEGORY>` data content. An element's data content is also a string.

You probably noticed that inside an attribute's value, we used the expression `'Scripting'` instead of `"Scripting"` for our string – we used single quotes not double quotes. This is because only the whole attribute's value can be enclosed by double quotes. Thus, any expression that needs to be enclosed, for example a string, is to be enclosed by single quotes, as in the following expression:

```
                    <xsl:if test="contains(CATEGORY/text(), 'Scripting')">
```

The next figure illustrates the HTML document produced using the transformation style sheet:


The book catalog listed in a table - Mi...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_if.htm
Go
Done
My Computer

Instant JavaScript	Scripting	1998-04-21	$49.34

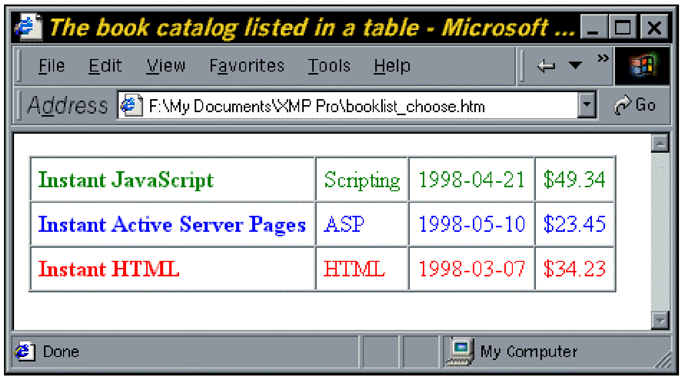
Sometimes we need a construct to perform some kind of action that is dependant on the item matched. For instance, in our previous example we filtered the original tree to transform only the `<ITEMS>` part of the Scripting category. In the next example, we will use a different transformation based on the category type.

In this example, we want to color code each table row with a different color, using a color for each category type. We can use a if/elseif pattern in the form of an `<xsl:choose>` element. This element is always used with its companion: the `<xsl:when>` element. Each condition is tested by the `<xsl:when>` construct, and more particularly by this element's test attribute:

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

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

<xsl:template name="DoTableBody">
   <th align="left"><xsl:apply-templates select="./TITLE"/></th>
   <td><xsl:apply-templates select="./CATEGORY" /></td>
   <td><xsl:apply-templates select="./RELEASE_DATE" /></td>
   <td><xsl:apply-templates select="./PRICE" /></td>
</xsl:template>

<xsl:template match="/">
   <html xmlns="http://www.w3.org/TR/xhtml1/strict">
      <head>
         <title>The book catalog listed in a table</title>
      </head>
      <body>
         <table border="1" cellspacing="0" cellpadding="5">
            <tbody>
               <xsl:for-each select="/BOOKLIST/ITEM">
                  <xsl:choose>
                     <xsl:when test="contains(CATEGORY/text(),'HTML')">
                        <tr style="color:red">
                           <xsl:call-template name="DoTableBody"/>
                        </tr>
                     </xsl:when>
                     <xsl:when test="contains(CATEGORY/text(),'Scripting')">
                        <tr style="color:green">
                           <xsl:call-template name="DoTableBody"/>
                        </tr>
                     </xsl:when>
                     <xsl:when test="contains(CATEGORY/text(),'ASP')">
                        <tr style="color:blue">
                           <xsl:call-template name="DoTableBody"/>
                        </tr>
                     </xsl:when>
                     <xsl:when test="contains(CATEGORY/text(),'JavaScript')">
                        <tr style="color:yellow">
                           <xsl:call-template name="DoTableBody"/>
                        </tr>
                     </xsl:when>
                  </xsl:choose>
               </xsl:for-each>
            </tbody>
         </table>
      </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

In the example above, we created an `if/elseif` construct with the `<xsl:choose>` element. Then each condition was tested with a `<xsl:when>` element. You probably noticed that we used the same expression as in the last example, but this time used it to check each case separately. In the first case, we check if the `<CATEGORY>` element's data content contains the string 'HTML'. If this is true, then we set the row style to use the color red. We then continue this pattern, checking for other category types.

Unlike in the last example, where we checked for a single condition, we check for several conditions. The output is shown below:


The book catalog listed in a table - Microsoft ...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_choose.htm
Go
Done
My Computer

Instant JavaScript	Scripting	1998-04-21	$49.34
Instant Active Server Pages	ASP	1998-05-10	$23.45
Instant HTML	HTML	1998-03-07	$34.23

An important fact to note is that because we used a template as a kind of subroutine, the simplified form cannot be used in this case. Because of this, we used the usual `<xsl:stylesheet>` construct instead.

Named templates

In the previous example we used a named template that had no parameters:

```
<xsl:template name="DoTableBody">
   <th align="left"><xsl:apply-templates select="./TITLE"/></th>
   <td><xsl:apply-templates select="./CATEGORY" /></td>
   <td><xsl:apply-templates select="./RELEASE_DATE" /></td>
   <td><xsl:apply-templates select="./PRICE" /></td>
</xsl:template>
```

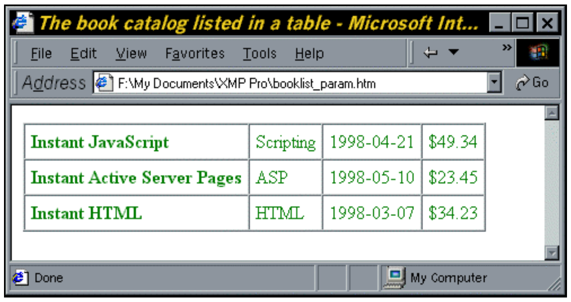
A named template can receive parameters. Let's say that in our example we want to pass, as a parameter, the row header alignment (which can be set to `left`, `right`, or `center`). To do so, we add the `<xsl:param . . . >` element to the named template, as shown in the following listing fragment:

```
<xsl:template name="DoTableBody">
   <xsl:param name="alignment">left</xsl:param>
   <xsl:param name="color">green</xsl:param>
   <tr style="color:{$color}">
      <th align="{$alignment}">
         <xsl:apply-templates select="./TITLE"/>
      </th>
      <td><xsl:apply-templates select="./CATEGORY" /></td>
      <td><xsl:apply-templates select="./RELEASE_DATE" /></td>
      <td><xsl:apply-templates select="./PRICE" /></td>
   </tr>
</xsl:template>
```

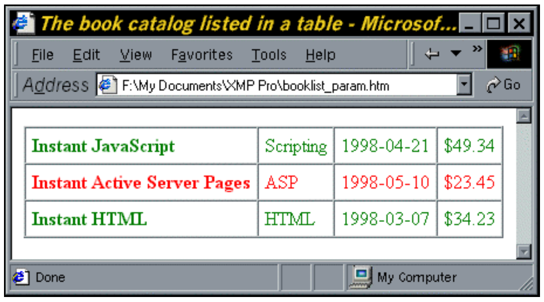
One parameter is named `alignment`, and its default value is `left`. The other one is named `color`, and its default value is `green`. These default values can be overridden by a `call-template`, with the parameters' values passed to the named template:

```
                                    <xsl:call-template name="DoTableBody">
                                       <xsl:with-param name="alignment">
                                          center
                                       </xsl:with-param>
                                       <xsl:with-param name="color">
                                          red
                                       </xsl:with-param>
                                    </xsl:call-template>
```

The following illustrations show the two different HTML documents created by a `call-template` construct. The first figure was created using `call-templates` without any parameters, and thus has both the `alignment` value set by default to `left` and the `color` value set to `green`. The other figure was created by changing the attributes for the `call-templates` construct in the ASP category to `center` and `red`.


The book catalog listed in a table - Microsoft Int...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_param.htm
Go
Done
My Computer

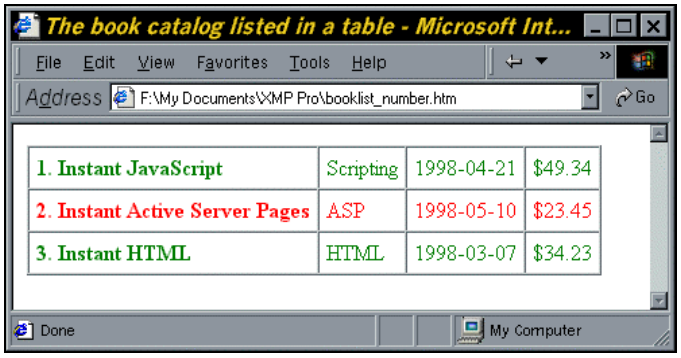
Instant JavaScript	Scripting	1998-04-21	$49.34
Instant Active Server Pages	ASP	1998-05-10	$23.45
Instant HTML	HTML	1998-03-07	$34.23


The book catalog listed in a table - Microsof...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_param.htm
Go
Done
My Computer

Instant JavaScript	Scripting	1998-04-21	$49.34
Instant Active Server Pages	ASP	1998-05-10	$23.45
Instant HTML	HTML	1998-03-07	$34.23

Numbering

Now, let's now add line numbering to our XSLT style sheet, so that we get the result illustrated below:


The book catalog listed in a table - Microsoft Int...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_number.htm
Go
Done
My Computer

1. Instant JavaScript	Scripting	1998-04-21	$49.34
2. Instant Active Server Pages	ASP	1998-05-10	$23.45
3. Instant HTML	HTML	1998-03-07	$34.23

To obtain this result we use the same style sheet as for the previous example, but this time we add the `<xsl:number ...>` construct. The construct's attributes, `value` and `format`, respectively indicate the value to insert in the output tree and the format used as a mask for the output. In our case we set the `value` attribute to the actual cursor position within the container element (the `<ITEM>` element). This is illustrated by the following code fragment:

```
<xsl:template name="DoTableBody">
   <xsl:param name="alignment">left</xsl:param>
   <xsl:param name="color">green</xsl:param>
   <tr style="color:{$color}">
      <th align="{$alignment}">
         <xsl:number value="position()" format="1. "/>
         <xsl:apply-templates select="./TITLE"/>
      </th>
      <td><xsl:apply-templates select="./CATEGORY" /></td>
      <td><xsl:apply-templates select="./RELEASE_DATE" /></td>
      <td><xsl:apply-templates select="./PRICE" /></td>
   </tr>
</xsl:template>
```

The `<xsl:number ... >` element can also use several other attributes, increasing its flexibility. For more details see the XSLT Recommendation at http://www.w3.org/TR/xslt.

The listing below is a combination of the last 5 examples, and is applying the following XSLT constructs to the original `Booklist` XML document:

- **Repetition** – using the `<xsl:for-each ...>` element
- **Sorting** – using the `<xsl:sort ...>` element
- **Conditional processing** – using both the `<xsl:if ...>` and `<xsl:choose ...>` elements.
- **Named templates** – using the `<xsl:template name ...>`, `<xsl:param ...>`, `<xsl:with-param ...>`, and `<xsl:call-template ...>` elements
- **Numbering** – using the `<xsl:number ...>` element

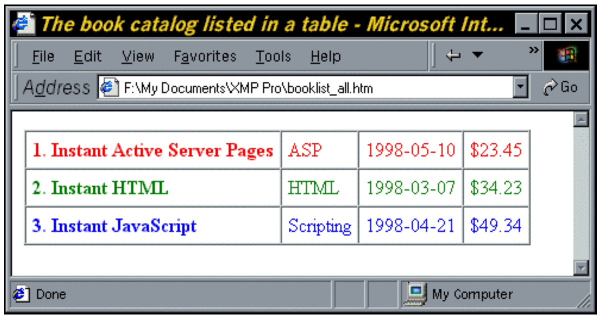
```
<?xml version="1.0"?>

<xsl:stylesheet xsl:version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:output method="html"/>

<xsl:template name="DoTableBody">
   <xsl:param name="alignment">left</xsl:param>
   <xsl:param name="color">green</xsl:param>
   <tr style="color:{$color}">
      <th align="{$alignment}">
         <xsl:number value="position()" format="1. "/>
         <xsl:apply-templates select="./TITLE"/>
      </th>
      <td><xsl:apply-templates select="./CATEGORY" /></td>
      <td><xsl:apply-templates select="./RELEASE_DATE" /></td>
      <td><xsl:apply-templates select="./PRICE" /></td>
   </tr>
</xsl:template>
```

```
   <xsl:template match="/">
      <html xmlns="http://www.w3.org/TR/xhtml1/strict">
         <head>
            <title>The book catalog listed in a table</title>
         </head>
         <body>
            <table border="1" cellspacing="0" cellpadding="5">
               <tbody>
                  <xsl:for-each select="/BOOKLIST/ITEM">
                     <xsl:sort select=".//CATEGORY"/>
                     <xsl:sort select=".//TITLE"/>
                     <xsl:choose>
                        <xsl:when test="contains(CATEGORY/text(),'HTML')">
                           <xsl:call-template name="DoTableBody"/>
                        </xsl:when>
                        <xsl:when test="contains(CATEGORY/text(),'Scripting')">
                           <xsl:call-template name="DoTableBody">
                              <xsl:with-param name="color">
                                 blue
                              </xsl:with-param>
                           </xsl:call-template>
                        </xsl:when>
                        <xsl:when test="contains(CATEGORY/text(),'ASP')">
                           <xsl:call-template name="DoTableBody">
                              <xsl:with-param name="alignment">
                                 center
                              </xsl:with-param>
                              <xsl:with-param name="color">
                                 red
                              </xsl:with-param>
                           </xsl:call-template>
                        </xsl:when>
                     </xsl:choose>
                  </xsl:for-each>
               </tbody>
            </table>
         </body>
      </html>
   </xsl:template>

   </xsl:stylesheet>
```

The following figure was obtained by:

❑ Transforming the original XML document with XT

❑ Rendering the transformation result (an HTML document) in Microsoft Internet explorer


The book catalog listed in a table - Microsoft Int...
File Edit View Favorites Tools Help
Address F:\My Documents\XMP Pro\booklist_all.htm
Go
Done
My Computer

1. Instant Active Server Pages	ASP	1998-05-10	$23.45
2. Instant HTML	HTML	1998-03-07	$34.23
3. Instant JavaScript	Scripting	1998-04-21	$49.34

Any XML browser that is fully compliant with the W3C recommendations will transform and render the document if the original XML document contains an `<xsl:stylesheet ...>` element.

Copying

We will now perform a different kind of manipulation on the `Booklist` XML document. We will keep the same document structure in the transformed document as in the original XML document, and keep the resultant formant as XML. We will simply sort the `<ITEM>` elements by their `<CODE>` value (data content). The following listing will do the job:

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

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

<xsl:output method="xml"/>

<xsl:template match="*">
   <xsl:copy>
      <xsl:apply-templates>
         <xsl:sort select=".//CODE"/>
      </xsl:apply-templates>
   </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Firstly, the only template rule is matched with any element node. Then, the `<xsl:copy>` element indicates to the XSLT engine to copy the element node to the result tree. We also tell the XSLT engine to sort the elements by the `<CODE>` element's value. The result is illustrated by the following XML document – the original document keeps its structure, but the `<ITEM>` elements are now sorted by the value of `<CODE>`:

```
<BOOKLIST>
   <ITEM>
      <CODE>16-041</CODE>
      <CATEGORY>HTML</CATEGORY>
      <RELEASE_DATE>1998-03-07</RELEASE_DATE>
      <TITLE>Instant HTML</TITLE>
      <PRICE>$34.23</PRICE>
   </ITEM>
   <ITEM>
      <CODE>16-048</CODE>
      <CATEGORY>Scripting</CATEGORY>
      <RELEASE_DATE>1998-04-21</RELEASE_DATE>
      <TITLE>Instant JavaScript</TITLE>
      <PRICE>$49.34</PRICE>
   </ITEM>
   <ITEM>
```

```
      <CODE>16-105</CODE>
      <CATEGORY>ASP</CATEGORY>
      <RELEASE_DATE>1998-05-10</RELEASE_DATE>
      <TITLE>Instant Active Server Pages</TITLE>
      <PRICE>$23.45</PRICE>
   </ITEM>
</BOOKLIST>
```

Transformation of an XML Document with the DOM

An XML document can also be transformed using the DOM, which is an interface to the grove representation of the document. However, using the DOM to transform an XML document can be a dangerous path to follow, since most DOM implementations contain a lot of proprietary constructs. To excuse the architects of these implementations, however, you should consider that this is mainly due to deficiencies in the DOM Recommendation. For instance, even the DOM2 recommendations do not specify how to load or save an XML document, so a DOM implementer will have to invent methods for these fundamental operations. The result of this is that most scripts you use to transform XML documents are probably not portable, as in many cases they will include some proprietary constructs.

Structural Transformations with the DOM

In order to compare the XSLT and DOM ways of structurally transforming XML documents, let's use the same example as we did in the *Structural Transformations* section earlier in the chapter.

To recap, we want to transform an XML document of the form:

```
<?xml version="1.0"?>
<BOOKLIST>
   <ITEM>
      <CODE>16-048</CODE>
      <CATEGORY>Scripting</CATEGORY>
      <RELEASE_DATE>1998-04-21</RELEASE_DATE>
      <TITLE>Instant JavaScript</TITLE>
      <PRICE>$49.34</PRICE>
   </ITEM>
   <ITEM>
      <CODE>16-105</CODE>
      <CATEGORY>ASP</CATEGORY>
      <RELEASE_DATE>1998-05-10</RELEASE_DATE>
      <TITLE>Instant Active Server Pages</TITLE>
      <PRICE>$23.45</PRICE>
   </ITEM>
   <ITEM>
      <CODE>16-041</CODE>
      <CATEGORY>HTML</CATEGORY>
      <RELEASE_DATE>1998-03-07</RELEASE_DATE>
      <TITLE>Instant HTML</TITLE>
      <PRICE>$34.23</PRICE>
   </ITEM>
</BOOKLIST>
```

into an XML document of the form:

```
<?xml version="1.0"?>
<BOOKLIST>
   <ITEM>
      <TITLE>Instant JavaScript</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: Scripting</CATEGORY>
         <CODE>(16-048)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>Release date: 1998-04-21</RELEASE_DATE>
         <PRICE>Price: $49.34</PRICE>
      </LISTING>
   </ITEM>
   <ITEM>
      <TITLE>Instant Active Server Pages</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: ASP</CATEGORY>
         <CODE>(16-105)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>release date: 1998-05-10</RELEASE_DATE>
         <PRICE>Price: $23.45</PRICE>
      </LISTING>
   </ITEM>
   <ITEM>
      <TITLE>Instant HTML</TITLE>
      <DESCRIPTION>
         <CATEGORY>Category: HTML</CATEGORY>
         <CODE>(16-041)</CODE>
      </DESCRIPTION>
      <LISTING>
         <RELEASE_DATE>release date: 1998-03-07</RELEASE_DATE>
         <PRICE>Price: $34.23</PRICE>
      </LISTING>
   </ITEM>
</BOOKLIST>
```

The next section will show how VBScript can modify the structure of the `Booklist` XML document.

VBScript Example

VBScript can be used to obtain the same result that we got using XSLT. The script in this section is not portable (for the reasons outlined in the last section) and can only run on a Windows platform. This is the main difference between using XSLT and using a script language with the DOM. Why choose VBScript? Simple – because 3 million developers can understand it.

The following script can be run through the Windows Script Host (WSH). Assuming you have WSH installed, this is as simple as saving this in a file called `Transform.vbs` and double clicking on it:

```
Set xmldoc = CreateObject("Microsoft.XMLDOM")
XMLDoc.load("C:\My Documents\XML Pro\Booklist.xml")

Set oItem = xmlDoc.getElementsByTagName("ITEM")
Set oTitle = xmlDoc.getElementsByTagName("TITLE")
Set oCode = xmlDoc.getElementsByTagName("CODE")
Set oPrice = xmlDoc.getElementsByTagName("PRICE")
```

```
   Set oRelease_Date = xmlDoc.getElementsByTagName("RELEASE_DATE")
   Set oCategory = xmlDoc.getElementsByTagName("CATEGORY")

   For i=0 To (oItem.length -1)
      Set oDescription = XMLDoc.createElement("DESCRIPTION")
      Set oTempDescription = oItem.item(i).appendChild(oDescription)
      oTempdescription.appendChild(oTitle.item(i))
      oTempDescription.appendChild(oCode.item(i))
      oTempDescription.appendChild(oCategory.item(i))
      Set oListing= XMLDoc.createElement("LISTING")
      Set oTempListing = oItem.item(i).appendChild(oListing)
      oTempListing.appendChild(oRelease_Date.item(i))
      oTempListing.appendChild(oPrice.item(i))
   Next

   XMLDoc.Save("sample.xml")
```

Let's examine this script in more detail. The first task is to create the DOM object with the `CreateObject()` method. We then load the source XML document into the DOM, parse it, and fill the internal tree structure – all with the `load()` method:

```
   Set xmldoc = CreateObject("Microsoft.XMLDOM")
   XMLDoc.load("C:\My Documents\XML Pro\Booklist.xml")
```

These last two lines are not part of the DOM recommendation, they are particular to the VBScript environment.

Because we are lucky enough not to have a complicated document structure, we can obtain the elements we need quite easily, using the `getElementsByTagName()` method. This would be more complicated if the document structure contained elements with the same name but located in different locations within the hierarchy. So, the next step is to obtain all the element objects needed for our transformation:

```
   Set oItem = xmlDoc.getElementsByTagName("ITEM")
   Set oTitle = xmlDoc.getElementsByTagName("TITLE")
   Set oCode = xmlDoc.getElementsByTagName("CODE")
   Set oPrice = xmlDoc.getElementsByTagName("PRICE")
   Set oRelease_Date = xmlDoc.getElementsByTagName("RELEASE_DATE")
   Set oCategory = xmlDoc.getElementsByTagName("CATEGORY")
```

Now we need to process any `<ITEM>` object element node contained in the booklist. To do this, we obtain the number of `<ITEM>` element node instances contained in the DOM, using the `length()` method. You would probably expect a `count()` method here, but the W3C recommendation uses the word `length()` to indicate an instance count. Notice here that the range of valid child node indices is `0` to `length-1` inclusive:

```
   For i=0 To (oItem.length -1)
```

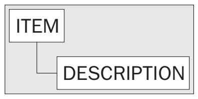
Because we have to add a new element as a child of the `<ITEM>` element, we create one with the object factory included in the DOM object:

```
      Set oDescription = XMLDoc.createElement("DESCRIPTION")
```

Then we append this new element as a child of the currently processed `<ITEM>` node. An element node object is returned, representing the `<DESCRIPTION>` element node:

```
        Set oTempDescription = oItem.item(i).appendChild(oDescription)
```

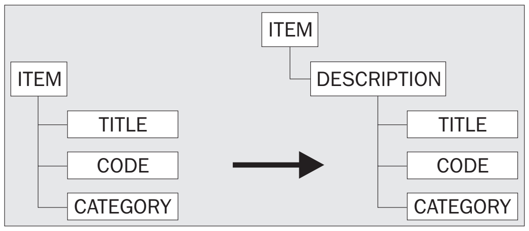
At this stage, we have modified the internal structure by adding a new `<DESCRIPTION>` node as child of the `<ITEM>` element node as illustrated below:



We then include the element nodes `<TITLE>`, `<CODE>`, and `<CATEGORY>` as children of the `<DESCRIPTION>` element node:

```
        oTempdescription.appendChild(oTitle.item(i))
        oTempDescription.appendChild(oCode.item(i))
        oTempDescription.appendChild(oCategory.item(i))
```

In fact, we are moving these nodes from their actual position to a new one, as illustrated in the following figure:



We then perform the same kind of operations for the new `<LISTING>` element node. We create it, insert it as child of the `<ITEM>` node element, and move the `<RELEASE_DATE>` and `<PRICE>` element nodes from their present position as children of the `<ITEM>` element node to their new position as children of the `<LISTING>` element node:

```
        Set oListing= XMLDoc.createElement("LISTING")
        Set oTempListing = oItem.item(i).appendChild(oListing)
        oTempListing.appendChild(oRelease_Date.item(i))
        oTempListing.appendChild(oPrice.item(i))
    Next
```

Finally, we save the transformed structure as an XML document:

```
    XMLDoc.Save("sample.xml")
```

Modifying an XSLT Document at Runtime

Up to now we have stayed within the boundaries of standards, and the previous examples can be executed with any XSLT processor that conforms to the W3C Recommendation. In this section, however, we'll use some proprietary Microsoft extensions to illustrate how XSLT can be used for user interactivity.

An XSL style sheet is activated by the inclusion of an `<xsl-stylesheet>` processing instruction within the XML document to be processed, or by interacting with it through a proprietary extension to the DOM. In all previous examples, we used the processing instruction as a link to the style sheet, but in this section, we'll use the Microsoft proprietary DOM extension to interface with the XSLT processor to show how XSLT can be used to sort the book list.

> *Several constructs used in this example are specific to Microsoft Internet Explorer version 5. Some constructs are simply outdated XSLT constructs and others are particular to this implementation and are not part of the W3C standard. Thus, the example listed will work only with the Microsoft Internet Explorer Version 5. Other XSLT engines such as XT will report errors.*

The XSLT script below can be used to render our `Booklist` XML document:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
   <html>
      <head>
         <title>The book catalog</title>
         <script>
            <xsl:comment>
               <![CDATA[
                  var xslStylesheet = null;
                  var xmlSource = null;
                  var attribNode = null;
                  function sort(field)
                  {
                     attribNode.value = field;
                     Booklist.innerHTML =
                       xmlSource.documentElement.transformNode(xslStylesheet);
                  }
               ]]>
            </xsl:comment>
         </script>
         <script for="window" event="onload">
            <xsl:comment>
               <![CDATA[
                  xslStylesheet = document.XSLDocument;
                  xmlSource = document.XMLDocument;
                  attribNode =
                     document.XSLDocument.selectSingleNode("//@order-by");
                  sort('TITLE');
               ]]>
            </xsl:comment>
         </script>
      </head>
      <body>
         <div id="Booklist"></div>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="BOOKLIST">
      <table border="0" frame="border" cellspacing="0" width="100%">
         <thead title="Alt-click sorts in descending order.">
            <tr style="background: brown;
                       color: white;
                       font-family: MS Sans Serif;
                       font-size:10pt">
               <th onclick="sort('TITLE')" width="33%" style="cursor:hand;">
                  <div>Title</div>
               </th>
               <th onclick="sort('CATEGORY')" width="20%" style="cursor:hand;">
                  <div>Category</div>
               </th>
               <th onclick="sort('CODE')" width="10%" style="cursor:hand;">
                  <div>Code</div>
               </th>
               <th onclick="sort('RELEASE_DATE')" style="cursor:hand;">
                  <div id="RELEASE_DATE">Release Date</div>
               </th>
               <th onclick="sort('PRICE')" width="10%" style="cursor:hand;">
                  <div>Price</div>
               </th>
            </tr>
         </thead>
         <tbody id="BOOKLIST_TABLE_BODY">
            <xsl:for-each select="//ITEM" order-by="//TITLE" >
               <tr>
                  <td><xsl:value-of select="TITLE"/></td>
                  <td><xsl:value-of select="CATEGORY"/></td>
                  <td><xsl:value-of select="CODE"/></td>
                  <td align="center"><xsl:value-of select="RELEASE_DATE"/></td>
                  <td><xsl:value-of select="PRICE"/></td>
               </tr>
            </xsl:for-each>
         </tbody>
      </table>
   </xsl:template>

   </xsl:stylesheet>
```

The first thing that the Microsoft Internet Explorer will do is to parse the XML and the XSL documents. The Microsoft parser creates a document model for both documents. Each of these documents can be accessed by an extended version of the DOM. Microsoft implemented the DOM with COM interfaces and added new functions. We can say that in some ways the Microsoft extended interface inherited from the W3C standard interfaces and added new properties or methods. It is these added methods, and not the standard W3C methods, that we will use in the example.

The elements implementing the run-time dynamic behavior are the `<script>` elements. At runtime the first script that is parsed and executed is the script not associated with any particular object. This script contains neither the `for` attribute nor the `event` attribute.

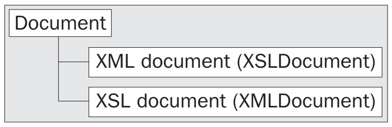
```
            <script>
               <xsl:comment>
                  <![CDATA[
                     var xslStylesheet = null;
                     var xmlSource = null;
                     var attribNode = null;
```

The script itself is enclosed by the `<xsl:comment>` element, which is translated (in the output tree) into the XML comment element. The script is written in JavaScript, and you probably noticed that we declared three objects and set them to a `null` value.

At runtime one of the first events to be fired is the `window.onload` event. We attached a script to this event:

```
<script for="window" event="onload">
   <xsl:comment>
      <![CDATA[
         xslStylesheet = document.XSLDocument;
         xmlSource = document.XMLDocument;
         attribNode =
            document.XSLDocument.selectSingleNode("//@order-by");
         sort('TITLE');
      ]]>
   </xsl:comment>
</script>
```

First, we obtain the XSL style sheet document from the document object. Then, we obtain the XML document from the document object. In the XML object hierarchy, the document object contains two extended DOMs: the XML extended DOM and the XSL extended DOM:



Both of these objects will useful to sort, transform, and display the XML document in Internet Explorer.

We also obtain the sort field object from the XSL tree. To get this object we ask the extended DOM to return the first object found that contains the `order-by` attribute. There is a single element containing this attribute: the `<xsl:for-each>` construct.

```
<xsl:for-each select="//ITEM" order-by="//TITLE" >
   <tr>
      <td><xsl:value-of select="TITLE"/></td>
      <td><xsl:value-of select="CATEGORY"/></td>
      <td><xsl:value-of select="CODE"/></td>
      <td><xsl:value-of select="RELEASE_DATE"/></td>
      <td><xsl:value-of select="PRICE"/></td>
   </tr>
</xsl:for-each>
```

We should mention that at the time of writing of this document, the latest specifications have changed the sorting mechanism, and this construct is no longer a standard one. The sort construct is now as we explained earlier. So, when Explorer becomes compatible with this specification, the sorting will be defined as:

```
            <xsl:for-each select="//ITEM">
               <xsl:sort select="TITLE"/>
               <tr>
                  <td><xsl:value-of select="TITLE"/></td>
                  <td><xsl:value-of select="CATEGORY"/></td>
                  <td><xsl:value-of select="CODE"/></td>
                  <td><xsl:value-of select="RELEASE_DATE"/></td>
                  <td><xsl:value-of select="PRICE"/></td>
               </tr>
            </xsl:for-each>
```

Thus, the JavaScript line obtaining the sort object becomes:

```
                      attribNode =
                        document.XSLDocument.selectSingleNode("xsl:sort/@select");
```

What is returned by `selectSingleNode()` is the attribute node, not the element node. Each element having one or more attributes is converted in the document tree to an element node with one or more attribute children. Thus, what the `attribNode` variable contains is the attribute node object.

Next, we sort and display the items by title order:

```
                      sort('TITLE');
```

Using the `sort()` function:

```
                      function sort(field)
                      {
                         attribNode.value = field;
                         Booklist.innerHTML =
                           xmlSource.documentElement.transformNode(xslStylesheet);
                      }
```

First, the attribute node object value is set with the '`TITLE`' string. This, in effect, modifies the extended DOM. So here, instead of modifying a XML document with XSL, we used the extended DOM – at runtime the XSL style sheet document can be modified with the extended DOM and an appropriate script.

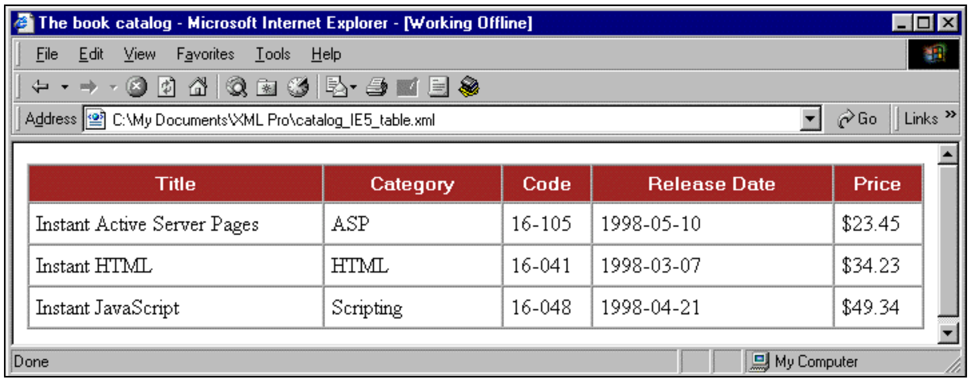
The second line requires more explanations. First, you may ask yourself where the `Booklist` object comes from. This object was created in the XSL script with the following construct:

```
         <body>
            <div id="Booklist"></div>
         </body>
```

As you can see, we created a uniquely identified object named `Booklist` when we created a `<DIV>` element. This is the HTML element that receives the result of the XML to HTML transformation. So, when the following expression is interpreted by the JavaScript interpreter:

```
                        Booklist.innerHTML =
                           xmlSource.documentElement.transformNode(xslStylesheet);
```

we call the `transformNode()` method on the `documentElement` object of the original XML document. This is the extended DOM method used to transform a XML document into HTML. This method takes as a parameter the XSL extended DOM that we stored in the `xslStylesheet` variable. The result is then stored in the `innerHTML` attribute of the `Booklist` object. This causes a refresh of the HTML document, and therefore a display refresh. The sorted table then appears on the screen:


The book catalog - Microsoft Internet Explorer - [Working Offline]
File Edit View Favorites Tools Help
Address C:\My Documents\XML Pro\catalog_IE5_table.xml
Go
Links
Done
My Computer

Title	Category	Code	Release Date	Price
Instant Active Server Pages	ASP	16-105	1998-05-10	$23.45
Instant HTML	HTML	16-041	1998-03-07	$34.23
Instant JavaScript	Scripting	16-048	1998-04-21	$49.34

The table headers have been set with a CSS style that indicates to the browser to display a small hand each time the cursor is on the table's headers. The user is accustomed to identifying a small hand with something that can be clicked on. Each column header is associated with the `sort()` function, which, in this case, acts as an event handler for the `onclick` event. If, for instance, the user clicks on the Price column, then the sort function is called with the `<onclick="sort('PRICE')" ...>` construct. When the `sort()` function receives the '`PRICE`' string as a parameter it sets this string as a value for the `attribNode` variable. This, in fact, changes the XSL style sheet. The modification is equivalent to writing the following construct:

```
            <xsl:for-each select="//ITEM" order-by="//PRICE" >
```

instead of:

```
            <xsl:for-each select="//ITEM" order-by="//TITLE" >
```

as was originally in the XSL style sheet. After the XSL extended DOM has been modified, we transform the original XML document again, using the modified XSL style sheet, and set the result of this transformation as input to the `innerHTML` attribute of the HTML `Booklist` object (named this way because of the `<DIV>` id).

So, because the XSL document is transformed into a tree structure (and because the extended DOM is the interface that Microsoft provides to this tree) it is possible to modify the XSL script with this interface. The modified XSL script can then perform a different transformation on the original document. Thus, to obtain different results based on the user interaction, the XSL style sheet can be modified from within script languages.

Comparing XSL Transformations with DOM Transformations

One of the key differences between the XSL and DOM transformation processes is that XSL is a **declarative** language rather than a **procedural** language. As such, XSL describes the state of the transformed document in relation to the original document. The DOM is an API that allows the manipulation of the tree structure.

The WSH VBScript we saw above used the DOM to achieve the same kind of transformation we did with an XSLT style sheet. But we can say that more elaborate transformation engines (conformant to the DOM1 or DOM2 specifications) are more limited than XSLT engines. This is mainly because the DOM1 and DOM2 Recommendations do not incorporate XPath expressions to reach a particular tree structure node. So, under certain circumstances, it can be a lot harder to use the DOM to transform an XML document than to use XSLT. If future editions of the DOM Recommendations include the capability to reach a particular node with an XPath expression, then using the DOM may be as easy and as powerful as using XSLT.

As we saw in the other DOM usage example, an XSLT style sheet can transform an XML document into HTML. The resultant HTML document can contain scripts that further manipulate the internal tree structure. Script procedures can be triggered by user actions, and these procedures can contain code using the DOM API to manipulate the XSLT document associated to the original XML document. This is what we did when we changed the value of some XSLT elements (the sorting value) when the user clicked on a table header to sort on this column category. In this example the DOM is used to change the value of an XML element's attributes, and because XSLT is itself an XML document, it can be modified with the DOM API. In this case, the DOM adds value to the XSLT transformation by providing sorting without having to include sorting code in the script.

Generally, we can say that in the actual state of the art, XSLT style sheets can be made more portable than scripts that use the DOM API. We saw that right at the beginning – the actual DOM1 and DOM2 Recommendations do not contain any constructs to load or save XML documents. Because of all this, it is better to use XSLT style sheets to perform transformations than it is to use scripts including DOM constructs.

Summary

In this chapter we have taken a look at transforming XML document structures. In particular we have spent a lot of time focusing on XSLT (XSL Transformations). This required knowledge of XPath and XPointers, which we meet in Chapter 8.

We have seen that there are several reasons why we may need to transform XML documents. These include:

- Transforming XML into a presentation language
- Translating between different vocabularies of XML
- Creating dynamic documents

XSLT is actually a huge topic, and hopefully this chapter will have got you used to the syntax of this particular specification. While it would be quite possible to write a whole book on the subject (indeed, look out for *XSLT Programmer's Reference* from Wrox Press, ISBN 1-861003-12-9), this should get you used to the overall functionality available, and give you a firm grounding in writing your own transformation style sheets.

The implementation that is available in Internet Explorer 5 was introduced before the XSLT specification was completed, so there are some differences, and some extensions. However, there are also other XSLT processors that you can use in your applications:

- XT – http://www.jclark.com/xml/xt.html
- SAXON – http://users.iclway.co.uk/mhkay/saxon