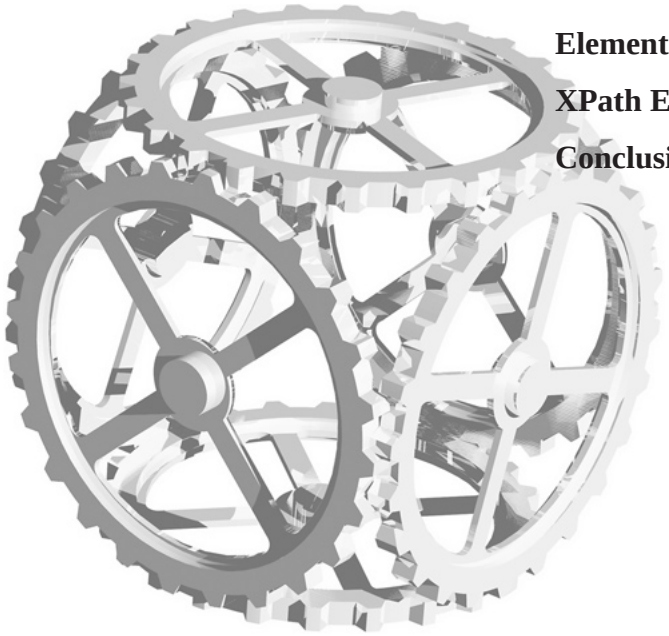
CHAPTER 5

XSLT

IN THIS CHAPTER:



In Chapter 2, we described XML as a metamarkup language. By itself, XML is fairly simple; it does nothing. But, in the last couple of chapters, we saw how parsers, *XML Schemas*, and the *Document Object Model* (*DOM*) offer tremendous functionality. You can use XML Schemas to add structure to your data, and then publish them for others to consume. You can use the DOM from within your applications to access and modify your XML. This chapter focuses on XSLT, which you can use to transform XML and produce output that can be displayed on the Web.

XSLT stands for *eXtensible Stylesheet Language Transformations.* The W3C describes XSLT as "a language for transforming XML documents into other XML documents." But XSLT can do more than that. Perhaps a better definition of XSLT requires a more generic slant: XSLT can transform the content and structure of an XML document into some other form.

While general and broad, this description hints at the power of XSLT. Another way to describe XSLT is to use an analogy: XSLT is to XML like SQL is to a database. Just as SQL can query and modify data, XSLT can query portions of an XML document and produce new content.

XSLT and XSL

XSLT is part of a much bigger family called *XSL* (*eXtensible Stylesheet Language*). XSL in its entirety represents two processes: tree transformation and formatting. XSLT focuses on tree transformation and can be used to process a source XML, producing a new XML document that can contain different nodes, different values, and different structures. The result tree, or *element and attribute tree*, may contain additional information like menus for Web applications. Or, the new tree could contain calculated values rather than the raw data. Or, maybe the tree has been reordered or filtered, creating a new, distinct tree.

The second process, formatting, is supported in XSL using the `fo` namespace, which stands for *formatting objects*. The `fo` namespace consists of elements and attributes to describe how the XML should be rendered. It's an extensive vocabulary that abstracts the process of displaying the content of the XML. The `fo` namespace provides the information necessary to present the XML, but leaves the actual rendering to the physical device. As described in Chapter 2, XML by itself separates content from presentation. The process of XSL formatting binds presentation to the XML content for specific rendering. This namespace is powerful, indeed, providing a rich language for producing high-quality output.

Saying that XSLT doesn't participate in the second process of XSL would be incorrect, however. On the contrary, XSLT can produce output that can be rendered.

One of the more popular uses of XSLT is to produce HTML so a source XML document can be viewed on the Web. XSLT can also produce plain text or even something as complex as a PDF file. Or, it can produce a document containing the formatting elements of the XSL `fo` namespace or other independent vocabulary, which can be used to render the XML directly.

But the `fo` namespace is more powerful than what is needed by standard browsers. This book is about building .NET Applications usable across the Web and, so, we focus on rendering XML as HTML. XSLT can accomplish this quite easily by itself or in conjunction with *Cascading Stylesheets* (*CSS* and *CSS2*). Thus, this chapter focuses on XSLT to produce HTML output.

XPath

When XSLT queries parts of an XML document, it makes use of another XML technology called *XPath*. XPath, like XML Schemas, the DOM, and XSLT, is a complex topic that could easily be the subject of an entire book. But, as we did with previous subject matters, we attempt to cover the essential details, providing a basis for the remaining chapters of this book. Therefore, we discuss XPath to the degree it's used by XSLT.

XPath was developed when the W3C realized significant overlap existed between the specification for selecting nodes using XSLT and the XPointer language. The W3C decided to create the XPath specification as the basis for both XSLT and XPointer.

The primary purpose of XPath is to select portions of an XML document using expressions to query for specific nodes within the document. But, it can also perform calculations, string manipulations, and evaluate expressions into boolean values.

How XSLT Works

XSLT documents are stored in stylesheets. The purpose of *stylesheets* is to transform XML. When a stylesheet transforms XML, a parser combines both the stylesheet and the XML to produce a new document. This document might be another XML document, or it could be HTML, or even plain text.

A good way to learn about XSLT is to jump in and look at some samples. Let's start with a simple one. The XML in Listing 5-1 contains information about the author Mark Twain and two of his book titles. The only new addition to this document is the Processing Instruction following the XML declaration. The Processing Instruction directs the parser to reference a stylesheet named

`Author.xsl`. The extension .xsl is standard for XSLT stylesheets. By including this instruction, however, the XML document won't parse correctly unless the parser can find and process the `Author.xsl` stylesheet.

Listing 5-1 *Simple XML Document*

L 5-1
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Author.xsl"?>
<author>
    <name>Mark Twain</name>
    <book>The Adventures of Huckleberry Finn</book>
    <book>Tom Sawyer</book>
</author>
```

Figure 5-2 contains the listing for the `Author.xsl` stylesheet.

Listing 5-2 *Simple XSLT Stylesheet*

L 5-2
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=" http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <body>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="author">
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="name">
        <H1>
            <xsl:value-of select="."/>
        </H1>
    </xsl:template>
    <xsl:template match="book">
        <li>
            <xsl:value-of select="."/>
        </li>
    </xsl:template>
</xsl:stylesheet>
```

Note how it looks just like XML, and it should. XSLT stylesheets are XML documents using the XSLT vocabulary.

When a parser transforms the XML in Listing 5-1, it loads the stylesheet in Listing 5-2 and produces the output in Listing 5-3.

Listing 5-3 *Stylesheet Output*

L 5-3
```
<html>
    <body>
        <H1>
            Mark Twain
        </H1>
        <li>The Adventures of Huckleberry Finn</li>
        <li>Tom Sawyer</li>
    </body>
</html>
```

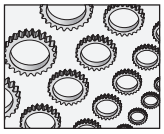
Figure 5-1 shows the output viewed from a browser:

Let's examine the process the parser used to generate this output line-by-line. With both the XML document and the stylesheet loaded, it begins by examining the stylesheet. A stylesheet is composed of templates and each template tells the parser how to process XML. A template contains a pattern the parser uses to match against either the entire XML document or portions of the document. The parser applies the template to each portion of the XML that matches the pattern.

The stylesheet in Listing 5-2 begins with a standard template that almost all stylesheets use; it matches against the root of the XML document. The attribute `match` contains the value /. This is an XPath expression that says find the beginning of the XML document. In Listing 5-1, the beginning of the document is the first line

Figure 5-1 *Browser displaying XML processed by the stylesheet*

following the XML declaration. The parser applies the template, running through the XSLT statements it contains. The template begins by instructing the parser to emit several HTML tags. Note, because the default namespace is set to the HTML schema, these tags don't contain a namespace prefix.



NOTE

The XML declaration isn't a node and, thus, isn't accessible using XPath.

The output now contains an `<html>` tag followed by the `<body>` tag. The next line in the template is an XSLT `apply-templates` statement. When this statement appears without the `select` attribute specifying an element to match against, the parser tries to find the first template in the stylesheet that matches an element within the XML document, beginning with the current element or node. Because the parser is at the beginning of the XML document, it searches the XSLT for any templates that match against the children of the first line after the XML declaration. Because the `xml-stylesheet` Processing Instruction doesn't match any of the templates defined in the stylesheet, the parser skips over this line. Next, the parser comes to the third line of Listing 5-1, the root document element, `author`.

Here, the parser does find a template that matches and it begins to apply this template. This template contains `author` as its pattern and contains only one instruction: the `apply-templates` statement. This tells the parser to construct a list of nodes from the children of the `author` element. It processes these children by iterating through this list, applying the templates in the stylesheet that match against each node.

The parser finds a matching template for the first child element, `name`. It applies this template first by emitting an `<H1>` tag. Next, the parser encounters a `value-of` statement. This statement is equivalent to a `select` statement in SQL and tells the parser to pull out from the current node the value as specified in its `select` expression. Just like the `match` attribute for the `template` statement, the `select` attribute consists of an XPath expression. The simple expression "." refers to the value of the current element. Because the current element is `name`, the parser emits its content, `Mark Twain`. It follows this with a terminating `</H1>` tag and returns to the parent template.

The parser, having processed the `name` element and its associated template, moves to the next node in its list, the first `book` element. Once again, the parser finds a template that matches this element. The first instruction in this template is to emit the `<li>` HTML tag. Next, it contains a similar line to the one in the template for

the `name` element. The parser emits the contents of the current node, adding `The Adventures of Huckleberry Finn` to the output. Finally, it adds a terminating `</li>` tag.

This process is then repeated for the second `book` element. Another pair of `<li>` and `</li>` tags are emitted, providing bookends for the second book title, `Tom Sawyer`.

The entire list of children has now been processed for the `author` element and the parser returns to the original template, continuing to process any remaining statements. The parser emits a terminating `</body>` tag and finishes with the terminating `</html>` tag. This is the end of the template and processing is complete.

The XSLT language is rich and powerful, therefore, no one right way exists to create a particular stylesheet. We could get exactly the same result using the stylesheet shown in Listing 5-4.

Listing 5-4 *Simple XSLT Stylesheet*

L 5-4
```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl"
    xmlns="http://www.w3.org/TR/REC-html40"
    result-ns="">
    <xsl:template match="/">
        <html>
            <body>
                <H1>
                    <xsl:value-of select="author/name"/>
                </H1>
                <xsl:for-each select="author/book">
                    <li><xsl:value-of select="."/></li>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

As you see in Listing 5-4, only one template is contained in this stylesheet. The parser processes it much the same as it did the first stylesheet. Because there are no `apply-template` instructions, it simply runs through each statement in the template and stops.

The parser begins by emitting the three HTML tags: `<html>`, `<body>`, and `<H1>`. Then, it comes to a `value-of` statement with a different XPath expression than in the other stylesheet, `author/name`. For now, assume this means, "find the value of the `name` element that is the child of the `author` element". The / is used to provide a sense of hierarchy in XPath expressions: it finds a `name` element that matches this expression and emits Mark Twain. Then it adds the terminating `</H1>` tag to the output.

The next line in Listing 5-4 is the XSLT `for-each` statement. This is a standard looping command common to most programming languages, which instructs the parser to loop through all the element children matching against the value of the `select` attribute. In this case, the XPath expression specifies `author/book`, which means the parser loops through each `book` element that is the child of the `author` element. For each iteration through the loop, it emits a `<li>` tag, followed by the value of the node being processed, and a terminating `</li>` tag.

Once the loop iterates through all the `book` elements, it finishes by terminating the `<body>` and `<html>` tags, respectively. The parser has now applied this template in its entirety, and processing is complete. Despite the differences, both stylesheets yield exactly the same output.

Elements of Transformation

The XSLT language is an application of XML, which means it has its own well-defined vocabulary of elements and attributes. This section is devoted to defining the use of these elements and their attributes. `xsl` is the standard namespace prefix used for the XSLT namespace, so we also use it. And, while the elements are listed in alphabetical order, it's important to note the `stylesheet` element is the root element of the XSLT vocabulary and, thus, must be the first element in a stylesheet. In addition, all the child elements of the `stylesheet` element are referred to as *top-level* elements.

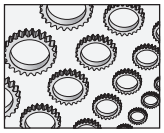
Not all the elements of XSLT are mentioned here. Instead, we focus on those elements used most often as a reference for examples later in this book. For a complete listing of the entire XSLT namespace, examine the specification at **http://www.w3.org/TR/xslt.html**.

xsl:apply-templates

The `apply-templates` element is used to apply a matching template, if one exists, to each child of the current node. Each child node is processed in turn, executing the appropriate template.

Attributes

Attribute	Usage	Value	Explanation
`mode`	Optional	`QName`	A name used to distinguish further which template to use for the list of nodes. A template must have a matching `mode` attribute to be considered for execution.
`select`	Optional	`XPath expression`	An expression used to select specific nodes. If omitted, all the children of the current node are selected.



NOTE

The datatype QName is a qualified name, which is either a simple NCName or an NCName with a namespace prefix.

Content

The `apply-templates` element may contain zero or more `sort` elements, and zero or more `with-param` elements.

Usage

The following snippet

L 5-5
```
<xsl:apply-templates select="//oranges">
    <xsl:sort select="price" data-type="number" order="ascending"/>
    <xsl:with-param name="nameOfFruit" select="string('Oranges')"/>
</xsl:apply-templates>
```

selects the list of nodes named `oranges`, and sorts the list based on the price child element in ascending order. For each node in the list, it passes to the matching template a value of `Oranges` to the template parameter `nameOfFruit`. Refer to the section of this chapter on the `template` element for a complete example.

xsl:attribute

The `attribute` element is used to add name/value attribute pairs to the output for the current element in the result tree. No other nodes can be added to this element in the result tree before these attribute nodes.

When XSLT is used to produce HTML output, this element is typically used to affect the values of style elements in the output. For example, the attribute element can display certain output text in different colors, depending on other values in the source XML document.

Attributes

Attribute	Usage	Value	Explanation
`name`	Mandatory	`Qname`	The name of the attribute to be generated by this statement.
`namespace`	Optional	`URI`	The namespace URI for the attribute being generated.

Content

The content of the `attribute` element contains the value of the attribute being generated. This value can be expressed simply as a textual value or by other XSLT elements that produce the value from the source XML.

Usage

For example, the following snippet of XSLT

L 5-6
```
<font>
    <xsl:attribute name="color">red</xsl:attribute>
    Hello World!
</font>
```

assigns the color red to the `<font>` element that surrounds the text `Hello World!`. The resulting HTML is

L 5-7
```
<font color="red">
    Hello World!
</font>
```

The value assigned by the attribute element can also be determined from the source XML. For example, the following snippet

L 5-8
```
<xsl:attribute name="color">
    <xsl:value-of select="textOutput/@desiredColor"/>
</xsl:attribute>
```

assigns the value of the `desiredColor` attribute of the `textOutput` element from the source XML to the `color` attribute of the `font` element, which is then emitted to the result tree. The `value-of` element, discussed later in this chapter, uses an XPath expression to select this value. The @symbol identifies the node `desiredColor` as an attribute of the element `textOutput`. The `/` symbol is used by XPath to ascribe hierarchy to the expression.

xsl:call-template

The `call-template` element is used to invoke a named template, much like invoking a procedure call in a typical programming language like Microsoft Visual Basic and C++.

Attributes

Attribute	Usage	Value	Explanation
`name`	Mandatory	`QName`	The name of the template to be invoked.

Content

The content of the `call-template` element may contain zero or more `with-param` elements.

Usage

For example, the following snippet of XSLT

L 5-9
```
<xsl:call-template name="displayFruit">
    <xsl:with-param name="nameOfFruit" select="string('Apples')"/>
</xsl:call-template>
```

invokes the template named `displayFruit`, passing the parameter `nameOfFruit` a value of `Apples`. Refer to the section of this chapter on the `template` element for a complete example.

xsl:choose

The `choose` element is used to select different instructions for processing the source XML based on defined tests. This is similar to the `Select Case` statement in Microsoft Visual Basic and the `switch` statement in C/C++.

The `choose` element makes use of the `when` and `otherwise` XSLT elements to set up different execution paths based on results of predefined tests.

Attributes

This element doesn't have any attributes.

Content

The `choose` element may contain one or more `when` elements. And, optionally, it may contain one `otherwise` element, but it must be the last child.

Usage

Given the following source XML

L5-10
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="C:\Goldilocks.xsl"?>
<root>
    <porridge temperature="234"/>
    <porridge temperature="60"/>
    <porridge temperature="100"/>
</root>
```

the following stylesheet

L5-11
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <b>Porridge</b>
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="porridge">
        <p>Temperature <xsl:value-of select="@temperature"/>:
        <font>
        <xsl:choose>
            <xsl:when test="@temperature &gt; 150">
                <xsl:attribute name="color">red</xsl:attribute>
                Too Hot
            </xsl:when>
            <xsl:when test="@temperature &lt; 70">
                <xsl:attribute name="color">blue</xsl:attribute>
```

```
                Too Cold
            </xsl:when>
            <xsl:otherwise>
                <xsl:attribute name="color">green</xsl:attribute>
                Just Right
            </xsl:otherwise>
        </xsl:choose>
        </font>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

produces the following HTML:

L 5-12

```
<b>Porridge</b>
<p>Temperature 234:
    <font color="red">Too Hot</font>
</p>
<p>Temperature 60:
    <font color="blue">Too Cold</font>
</p>
<p>Temperature 100:
    <font color="green">Just Right</font>
</p>
```

When you walk through this example, you see the template matching the `porridge` elements contains a `choose` element, which contains two `when` elements and an `otherwise` element. The two `when` elements test the value of the temperature attribute of the current node. If the value of the attribute is greater than `150`, then two instructions are executed: the `attribute` element assigns the `font` element's `color` attribute to a value of `red`, and the text `Too Hot` is written to the output. If the value is less than `70`, then the `font` element's `color` attribute is set to `blue`, and the text `Too Cold` is sent to the output. If the value doesn't fall into either category, then the `otherwise` element is executed, which sets the `font` element's `color` attribute to `green`, and emits the text `Just Right`.

xsl:decimal-format

The `decimal-format` element is a top-level element used to control the way the XPath `format-number()` function formats numbers into strings. This element neither affects the way the `number` and `value-of` elements format numbers for output nor the XPath `string()` function.

Attributes

Attribute	Usage	Value	Explanation
`decimal-separator`	Optional	`character`	Specifies the character used to separate the integral whole number from its fractional counterpart. The default is "`.`".
`digit`	Optional	`character`	Specifies the character used to indicate digits in the format pattern. The default is #.
`grouping-separator`	Optional	`character`	Specifies the character used to separate numbers exceeding thousands, millions, and so forth. The default is `,`.
`infinity`	Optional	`string`	Specifies the string that represents a value of infinity. The default is `Infinity`.
`minus-sign`	Optional	`character`	Specifies the character used to indicate negative numbers. The default is `-`.
`name`	Optional	`QName`	Binds a name to this decimal format. If omitted, this format becomes the default.
`NaN`	Optional	`string`	Specifies the string used to represent a nonnumerical value, NaN, or "not-a-number." The default is `NaN`.
`pattern-separator`	Optional	`character`	Specifies the character used to separate the pattern used for positive values from the pattern used for negative numbers. The default is `;`.
`percent`	Optional	`character`	Specifies the character used to represent the percent sign. The default is %.
`per-mille`	Optional	`character`	Specifies the character used to represent the per-mille sign or the sign used to represent the percentage of thousandths. The default is the Unicode character `#x2030`, ‰.
`zero-digit`	Optional	`character`	Specifies the character used in the format pattern to indicate leading zeroes. The default is `0`.

Content

The `decimal-format` element is an empty element, containing no other content or child elements.

Usage

Here's an example of using this element in a stylesheet:

```
L5-13   <xsl:decimal-format
            name="american-standard"
            decimal-separator="."
            grouping-separator=","/>
```

Later in the same stylesheet, the following line

```
L5-14   <xsl:value-of
            select="format-number(345343.456, '#,##0.00',
        'american-standard')"/>
```

would yield this output

```
L5-15   345,346.46
```

while this line

```
L5-16   <xsl:value-of
            select="format-number(3.456, '#,##0.00', 'american-standard')"/>
```

would yield this output

```
L5-17   3.46
```

xsl:for-each

The `for-each` element provides looping functionality similar to the standard for-loops found in programming languages like Microsoft Visual Basic and C++. This instruction is often used when the source XML is highly consistent. When the source is not so coherent, the `template` element can be used as an alternative.

Attributes

Attribute	Usage	Value	Explanation
`select`	Mandatory	`XPath Expression`	Selects the set of nodes to be processed by the `for-each` element.

Content

The `for-each` element may contain zero or more `sort` elements, followed by output elements.

Usage

Here's an example of using this element in a stylesheet:

L5-18
```
<xsl:for-each select="//oranges">
    <xsl:sort select="price" order="ascending" data-type="number"/>
    Price: <xsl:value-of select="price"/>
</xsl:for-each>
```

This sample generates a list of nodes containing all the `oranges` elements from the source XML. This list is sorted by the child element `price`. Then the parser iterates through each node in the list emitting the value of each `price` element.

xsl:if

The `if` element provides `if-then` functionality, similar to `if-then` clauses found in programming languages like Microsoft Visual Basic and C++. However, unlike these languages, XSLT doesn't provide an `else` clause.

Attributes

Attribute	Usage	Value	Explanation
`Test`	Mandatory	`XPath Expression`	Provides the boolean expression to be tested.

Content

The `if` element may contain a set of output elements.

Usage

Here's an example of using this element in a stylesheet:

L 5-19
```
<font>
    <xsl:if test="porridge[@temperature > 150]">
        <xsl:attribute name="color">red</xsl:attribute>Too Hot
    </xsl:if>
</font>
```

This sample is similar to the example for the choose element but, unlike that example, there aren't any alternative paths. Either the expression proves to be true and the output is generated, or it's skipped altogether.

xsl:import

The `import` element is a top-level element that must come before any other children of the `stylesheet` element. It's used to import one stylesheet into another. However, the rules and definitions of the importing stylesheet take on a higher precedence than those of the stylesheet being imported.

Attributes

Attribute	Usage	Value	Explanation
`href`	Mandatory	`URI`	Contains the URI reference to the stylesheet being imported.

Content

The `decimal-format` element is an empty element, containing no other content or child elements.

Usage

L 5-20
```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:import href="C:\…\AnotherStylesheet.xsl"/>
</xsl:stylesheet>
```

xsl:include

The `include` element is a top-level element used to include one stylesheet into another. Unlike the `import` element, the rules and definitions of the included stylesheet take on the same precedence as those of the including stylesheet.

Attributes

Attribute	Usage	Value	Explanation
`href`	Mandatory	`URI`	Contains the URI reference to the stylesheet being included.

Content

The `decimal-format` element is an empty element, containing no other content or child elements.

Usage

L 5-21

```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:include href="C:\…\AnotherStylesheet.xsl"/>
</xsl:stylesheet>
```

xsl:otherwise

The `otherwise` element is used within a `choose` element, much like a final `else` or `default` statement in modern programming languages. If none of the test conditions in any of the other `when` elements are satisfied, then the instructions in the `otherwise` element are executed by default.

Attributes

The `otherwise` element has no attributes.

Content

This element may contain output elements.

Usage

Refer to the section of this chapter on the `choose` element for an example on how the `otherwise` element is used.

xsl:output

The `output` element is a top-level element used to control the format of the output generated by the stylesheet. This element is used during the second stage of processing, when the result tree from the first stage is written to an output stream or file.

Attributes

Attribute	Usage	Value	Explanation
`cdata-section-elements`	Optional	`Whitespace separated list of QNames`	Lists the elements whose textual content is to be output as `CDATA` sections.
`doctype-public`	Optional	`string`	Specifies the public identifier used in the `document type declaration` (if any) in the output.
`doctype-system`	Optional	`string`	Specifies the system identifier used in the `document type declaration` (if any) in the output.
`encoding`	Optional	`string`	Specifies the character encoding used to encode sequences of characters.
`indent`	Optional	`"yes" | "no"`	Determines whether the output should be indented t o highlight its hierarchical structure.
`media-type`	Optional	`string`	Binds a media-type to the output, typically MIME.

Content

The `decimal-format` element is an empty element, containing no other content or child elements.

Usage

L 5-22
```
<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
```

xsl:param

The `param` element can be used either as a top-level element to create a global parameter or within a `template` element to define a local parameter. If used as a child of a `template` element, then it must be the first child element.

Attributes

Attribute	Usage	Value	Explanation
`name`	Mandatory	`QName`	Assigns a name to the parameter.
`select`	Optional	`XPath Expression`	Assigns a default value for the parameter. When used within a `template`, the caller can assign a different value, overriding this default value.

Content

If the `select` attribute is supplied, this element is empty. Otherwise, this element may contain elements that define a default value for the parameter.

Usage

This snippet

L 5-23
```
<xsl:param name="nameOfFruit" select="NameOfFruit"/>
```

defines a parameter named `nameOfFruit` and provides a default value of `NameOfFruit`.

xsl:preserve-space

The `preserve-space` element is a top-level element used to control the way whitespace is handled (see `strip-space`).

Attributes

Attribute	Usage	Value	Explanation
`elements`	Mandatory	`NameTest`	Lists the elements whose whitespace-only text nodes are to be preserved during processing.

Content

The `decimal-format` element is an empty element, containing no other content or child elements.

Usage

This element instructs the parser to maintain whitespace-only text nodes occurring as children of the elements specified in the `elements` attribute. So, the following

L 5-24
```
<xsl:preserve-space elements="apples oranges"/>
```

preserves whitespace-only text nodes for the elements `apples` and `oranges`.

xsl:stylesheet

The root element for XSLT is `stylesheet`; therefore, it must be the first element of every XSLT document.

Attributes

Attribute	Usage	Value	Explanation
`extension-element-prefixes`	Optional	`NCNames`	Lists the namespaces used by this stylesheet that provide extension elements and extension functions.
`exclude-result-prefixes`	Optional	`NCNames`	Lists the namespaces not to be copied into the result tree.
`id`	Optional	`Name`	Used to identify a stylesheet so it can be referenced by another XML document.
`version`	Mandatory	`Number`	Identifies the version of XSLT used by the stylesheet.

Content

The `stylesheet` element can contain several types of child elements, referred to as *top-level* elements. Of these, we delve into the following elements:

Child Element	Description
`attribute-set`	Used to define a named set of attribute name/value pairs, which can be applied to an output element. Uses the `attribute` element to create the sets of attributes. Refer to the XSLT specification, section 7.1.4, for more information.
`decimal-format`	Specifies the characters and symbols used by the `format-number()` function to convert numbers into strings.
`import`	Used for importing other stylesheets.
`include`	Used for including other stylesheets.
`key`	Defines a named key for use with the XPath function `key()`, allowing a stylesheet to index nodes and their values. Refer to the XSLT specification, section 12.2, for more information.
`namespace-alias`	Aliases a namespace in the stylesheet, mapping it to another namespace used in the output. Refer to the XSLT specification, section 7.1.1, for more information.
`output`	Defines the output of the resulting file during the second stage of transformation.
`param`	Defines a global parameter (or local parameter when used inside the `template` element) in a stylesheet and assigns it a default value. This is different from the `variable` element because templates can be called passing in a value for the `param` element.
`preserve-space`	Defines the way whitespace is preserved in the result tree.
`strip-space`	Defines the way spaces are stripped out of the source XML.
`template`	Defines rules for producing output against some specified matching portion of the source XML.
`variable`	Defines a global variable (or local variable when used inside the `template` element) in a stylesheet and assigns it a value.

Usage

The following snippet

L 5-25
```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

demonstrates the most common usage of this statement, specifying the version and the schema namespace for the XSL vocabulary.

xsl:template

The `template` element is a top-level element and is essential for any meaningful stylesheet. The `template` element is used to define how portions of an XML document should be transformed. Essentially, transformation begins with this element.

Attributes

Attribute	Usage	Value	Explanation
`match`	Optional	`XPath Expression`	Used to generate a list of nodes to be processed by this template. If this attribute isn't present, then there must be a `name` attribute.
`mode`	Optional	`QName`	The mode of the template. When `apply-templates` is used to process a list of nodes, only those templates with a matching mode are invoked.
`Name`	Optional	`QName`	The name of this template. If this attribute is missing, then a `match` attribute must exist.
`Priority`	Optional	`Number`	A number used to prioritize which template should be invoked when more than one template matches a given node.

Content

Child Element	Description
`Param`	When used within a template, defines a local parameter. Templates can be called passing in a value for the `param` element.

The rest of the content of the `template` element are the instructions for generating the result tree.

Usage

As an example, say you have an XML document that contains information about some fruit purchased recently. The document contains the type of fruit, its place of origin,

where it was purchased, the quantity, and the price of the purchase. For example, the XML in Listing 5-5 is processed using the stylesheet shown in Listing 5-6:

Listing 5-5 *Apples and Oranges XML Document*

```
L 5-26  <?xml version="1.0" encoding="UTF-8"?>
        <?xml-stylesheet type="text/xsl" href="C:\…\ApplesOranges.xsl"?>
        <root>
            <apples>
                <origin>California</origin>
                <quantity>1000</quantity>
                <price>35</price>
            </apples>
            <oranges>
                <origin>California</origin>
                <quantity>2000</quantity>
                <price>22</price>
            </oranges>
            <oranges>
                <origin>Florida</origin>
                <quantity>1500</quantity>
                <price>21</price>
            </oranges>
            <apples>
                <origin>Florida</origin>
                <quantity>1200</quantity>
                <price>26</price>
            </apples>
        </root>
```

Listing 5-6 *Apples and Oranges Stylesheet*

```
L 5-27  <?xml version="1.0" encoding="UTF-8"?>
        <xsl:stylesheet
            version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
            <xsl:template match="/">
                <html>
```

```
        <body>
        <xsl:apply-templates select="//apples">
            <xsl:sort select="price" data-type="number" order="ascending"/>
        </xsl:apply-templates>
        <xsl:apply-templates select="//oranges">
            <xsl:sort select="price" data-type="number" order="ascending"/>
        </xsl:apply-templates>
        </body>
        </html>
    </xsl:template>
    <xsl:template match="apples">
        <xsl:call-template name="displayFruit">
            <xsl:with-param name="nameOfFruit" select="string('Apples')"/>
        </xsl:call-template>
    </xsl:template>
    <xsl:template match="oranges">
        <xsl:call-template name="displayFruit">
            <xsl:with-param name="nameOfFruit" select="string('Oranges')"/>
        </xsl:call-template>
    </xsl:template>
    <xsl:template name="displayFruit">
        <xsl:param name="nameOfFruit" select="NameOfFruit"/>
            <p>
                <b><xsl:value-of select="$nameOfFruit"/>:</b>
                <ul><li>Origin: <xsl:value-of select="origin"/></li></ul>
                <ul><li>Quantity: <xsl:value-of select="quantity"/></li></ul>
                <ul><li>Price: <xsl:value-of select="price"/></li></ul>
            </p>
    </xsl:template>
</xsl:stylesheet>
```

The stylesheet in Listing 5-6 demonstrates the way many of the XSLT elements we have discussed can be used. The main purpose of this stylesheet is to group like fruit together, and then to sort them in ascending order by price. This is akin to the GROUP BY and ORDER BY statements in SQL.

As usual, the stylesheet begins with a `template` element that uses the pattern / to match against the first line of the XML document. The first instruction of this template is to apply any and all templates against the list of elements matching the name `apples`.

In addition, before the list of nodes is passed off to any matching templates, the list is sorted using the `sort` element. The `select` attribute indicates which element or

attribute to sort on, in this case, the `price` element. The `data-type` attribute, with a value of `number`, instructs the parser that the value being sorted is a number. And the `order` attribute, set to `ascending`, says to sort in ascending order. So now, you've accomplished an `ORDER BY` SQL instruction. The sorted listed is passed to any and all matching templates.

One template's `match` attribute contains the pattern `apples`. Therefore, this template is used for the entire sorted list. Despite how the XML document was organized, XSLT is clearly restructuring the result tree to have the list of elements named `apples` come first. Essentially, this is a `GROUP BY` statement with `apples` as the first token in the list.

The instructions inside this template call another template by the name of `displayFruit`, passing it a value of `Apples` for the parameter `nameOfFruit`. The `displayFruit` template now produces the actual output. It begins by emitting two HTML tags: `<p>` and `<b>`. Then, using the `value-of` element, it writes to the output the value of the parameter `nameOfFruit`, which, in this case is `Apples`. After terminating with a `</b>` tag, the template emits `<ul>` and `<li>` tags to create a bulleted, indented line of text, beginning with `Origin:`. It uses a `value-of` element to select the value of the `origin` child element of the current `apple` element. The template terminates the line with the `</li>` and `</ul>` tags. It does the same thing for the `quantity` and `price` child elements, using the `value-of` element to write their respective values to the output, bracketed them with the `<ul><li>` and `</li></ul>` HTML tags, and the text `Quantity:` and `Price:`. The template finishes with a terminating `</p>` tag. It does this for each `apples` element in the sorted list of nodes.

Once this template has run its course, it yields to the prior template, which has also completed all its instructions. So now focus is returned to the original template. There, the next instruction to be executed looks just like the first one, except it operates on all the `oranges` elements.

Again, the list of nodes is sorted by the `price` element, but this list is passed to the template with the `match` attribute set to `oranges`. This results in a result tree with the `oranges` elements grouped after the `apples` elements. Just as the template did for the `apples` elements, this template calls the `displayFruit` template, passing the `nameOfFruit` parameter a value of `Oranges`.

The `displayFruit` template executes just as it did before, but now on the list of `oranges` elements, which have been sorted in ascending order by `price`. First, it emits a `<p>` tag. Then the template surrounds the parameter value of `Oranges` with

 and tags. And, then, it creates three bulleted, indented lines containing the values of the `origin`, `quantity,` and `price` elements, respectively.

Once the template has done this for each node in the list, execution is ultimately returned to the original template and processing is complete. This stylesheet produces the output in Listing 5-7:

Listing 5-7 *Apples and Oranges Output*

L 5-28
```
<html>
    <body>
        <p>
            <b>Apples:</b>
            <ul><li>Origin: Florida</li></ul>
            <ul><li>Quantity: 1200</li></ul>
            <ul><li>Price: 26</li></ul>
        </p>
        <p>
            <b>Apples:</b>
            <ul><li>Origin: California</li></ul>
            <ul><li>Quantity: 1000</li></ul>
            <ul><li>Price: 35</li></ul>
        </p>
        <p>
            <b>Oranges:</b>
            <ul><li>Origin: Florida</li></ul>
            <ul><li>Quantity: 1500</li></ul>
            <ul><li>Price: 21</li></ul>
        </p>
        <p>
            <b>Oranges:</b>
            <ul><li>Origin: California</li></ul>
            <ul><li>Quantity: 2000</li></ul>
            <ul><li>Price: 22</li></ul>
        </p>
    </body>
</html>
```

And Figure 5-2 shows how the HTML from Listing 5-7 is displayed in a browser.

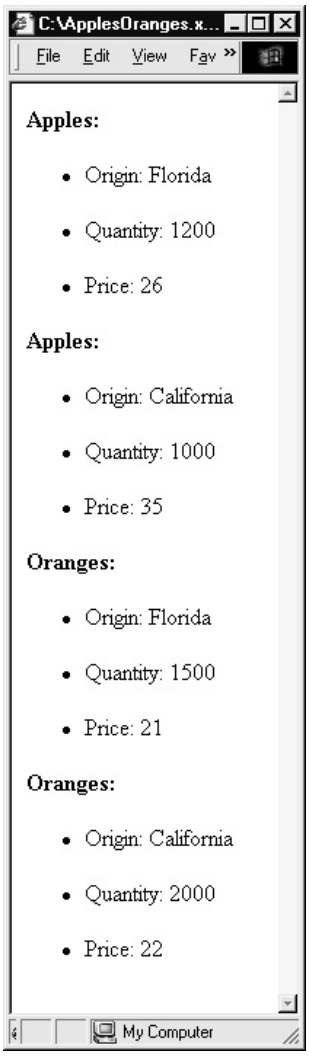


Figure 5-2 *Browser view of apples and oranges*

xsl:value-of

The `value-of` element is used to send a value from the specified element or attribute to the result tree.

Attributes

Attribute	Usage	Value	Explanation
`disable-output-escaping`	Optional	`"yes" | "no"`	A value of `yes` instructs the parser to allow special characters such as < to be output rather than using the XML escape form `<`. The default value is `no`.
`select`	Mandatory	`XPath Expression`	Selects the value to be sent to the output.

Content

The element is always empty.

Usage

The following snippet

L 5-29
```
<xsl:value-of select="origin"/>
```

selects the value of the `origin` element for output.

xsl:variable

The `variable` element can be used either as a top-level element to create a global variable or within a `template` element to define a local variable.

Attributes

Attribute	Usage	Value	Explanation
`name`	Mandatory	`QName`	Assigns a name to the variable.
`Select`	Optional	`XPath Expression`	Assigns an initial value for the variable. If this is omitted, then the contents of this element define the value of the variable.

Content

If the `select` attribute is supplied, this element is empty. Otherwise, the contents of this element are used to define the value of the variable.

Usage

The following snippet

L 5-30
```
<xsl:variable name="anotherVariable">
    <xsl:value-of select="fruit/price"/>
</xsl:variable>
```

creates a variable named `anotherVariable`, and assigns it the value of the `price` element. The XPath expression used in the `select` attribute finds the value of the price element by examining the current element and selecting the first `fruit` child element. From there, it then finds the first `price` child element.

xsl:when

The `when` element is used within a `choose` element body to evaluate a particular condition. If the condition tests true, then the output instructions it contains are executed.

Attributes

Attribute	Usage	Value	Explanation
`Test`	Mandatory	`XPath Expression`	Defines the test expression that resolves to a boolean value.

Content

The `when` element contains output elements.

Usage

Refer to the `choose` element for an example of how this element is used.

xsl:with-param

The `with-param` element is used within either the `apply-templates` or `call-template` element to set the value of local parameters used inside a template.

Attributes

Attribute	Usage	Value	Explanation
`Name`	Mandatory	`QName`	The name of the parameter that will be assigned a value.
`Select`	Optional	`XPath Expression`	Assigns the value for the parameter.

Content

If the `select` attribute is supplied, this element should be empty. Otherwise, the body contains the value to be assigned to the parameter.

Usage

The following snippet

L 5-31
```
<xsl:call-template name="displayFruit">
    <xsl:with-param name="nameOfFruit" select="string('Apples')"/>
</xsl:call-template>
```

calls the `displayFruit` template, first passing the template's `nameOfFruit` parameter a value of `Apples`.

XPath Expressions

In describing the XSLT elements, we used examples containing XPath expressions to select values for output, to calculate results, and to test conditions. XPath allows XSLT to navigate the source XML quickly using a string that describes the navigation path. It also provides functions that act on nodes to generate values or test whether a particular condition is true.

As with XSLT, an example is the best way to begin understanding how XPath expressions work within a stylesheet. The following XML is similar to the apples and oranges example in Listing 5-5 used to describe the `template` element. In Listing 5-8, though, the XML contains only `apples` elements:

Listing 5-8 *Just Apples XML Document*

L 5-32
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="C:\…\JustApples.xsl"?>
<root>
```

```
    <apples>
        <origin>California</origin>
        <quantity>1000</quantity>
        <price>35</price>
    </apples>
    <apples>
        <origin>Florida</origin>
        <quantity>600</quantity>
        <price>24</price>
    </apples>
    <apples>
        <origin>Florida</origin>
        <quantity>1000</quantity>
        <price>30</price>
    </apples>
    <apples>
        <origin>California</origin>
        <quantity>400</quantity>
        <price>20</price>
    </apples>
    <apples>
        <origin>California</origin>
        <quantity>100</quantity>
        <price>28</price>
    </apples>
    <apples>
        <origin>Florida</origin>
        <quantity>1200</quantity>
        <price>26</price>
    </apples>
</root>
```

The stylesheet in Listing 5-9 operates on this XML, trying to calculate certain values for those `apples` elements originating from the same state:

Listing 5-9 *Just Apples Stylesheet*

```
L 5-33  <?xml version="1.0" encoding="UTF-8"?>
        <xsl:stylesheet
```

```
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
        <body>
        <xsl:variable
            name="origins"
            select="//origin[not(. = preceding::origin)]"/>
        <xsl:for-each select="$origins">
            <xsl:variable
                name="sameOrigin"
                select="//apples[origin = current()]"/>
            <p>
                <b><xsl:value-of select="."/></b>
                <ul><li>
                    Quantity:
                    <xsl:value-of select="sum($sameOrigin/quantity)"/>
                </li></ul>
                <xsl:apply-templates select="$sameOrigin">
                    <xsl:sort select="price" order="ascending"/>
                </xsl:apply-templates>
            </p>
        </xsl:for-each>
        </body>
        </html>
    </xsl:template>
    <xsl:template match="apples">
        <xsl:if test="position() = last()">
            <ul><li>Max: <xsl:value-of
select="current()/price"/></li></ul>
        </xsl:if>
        <xsl:if test="position() = 1">
            <ul><li>Min: <xsl:value-of
select="current()/price"/></li></ul>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

When the XML in Listing 5-8 is combined with the stylesheet in Listing 5-9, the HTML in Listing 5-10 is generated.

Listing 5-10 *Just Apples Output*

L 5-34
```
<html>
    <body>
        <p>
            <b>California</b>
            <ul><li>Quantity: 1500</li></ul>
            <ul><li>Min: 20</li></ul>
            <ul><li>Max: 35</li></ul>
        </p>
        <p>
            <b>Florida</b>
            <ul><li>Quantity: 2800</li></ul>
            <ul><li>Min: 24</li></ul>
            <ul><li>Max: 30</li></ul>
        </p>
    </body>
</html>
```

And Figure 5-3 shows how the HTML from Listing 5-10 is displayed in a browser.

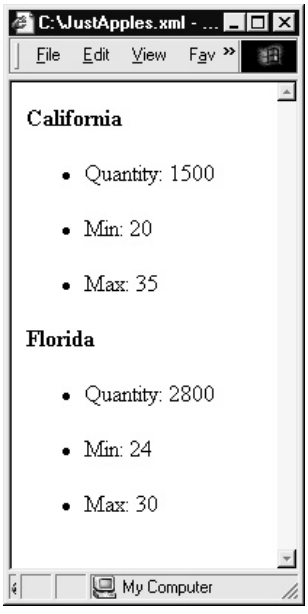
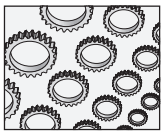


Figure 5-3 *Browser view of just apples*

Let's walk though this example, line by line. The stylesheet in Listing 5-9 begins, as usual, with a template that matches against the beginning of the source XML using the expression `/`. This special character, `/`, can be used in several ways to denote hierarchy within an XML document and to navigate up or down the XML node tree. Table 5-1 details several different ways it can be used.



NOTE

An axis is a direction of navigation through the XML tree from a starting point. This is discussed later in this chapter.

Back to the stylesheet in Listing 5-9, the first line in the template creates a variable named `origins`. It assigns this variable a value based on the XPath

Expression	Meaning
`/`	Matches the first node of the source XML, including any comments or Processing Instructions, excluding the XML declaration.
`//apples`	The `AbrreviatedAbsoluteLocationPath` expression, `//`, indicates the expression begins at the document root element. It selects all the `apples` elements in the document.
`//apples[@type='red']`	Finds all the `apples` elements in the document with an attribute named `type` having a value of `red`. The `@` symbol is used to indicate the search continues along the attribute axis.
`//oranges[origin='Florida']`	Finds all the `oranges` elements in the document whose child element `origin` contains the value `Florida`.
`.//origin`	Selects all the `origin` elements that are children of the current node context.
`//*/*`	Selects all elements in the document that have an element as a parent. Basically, this selects all elements in the document, except those that are the immediate children of the root node. The `*` matches against any element.
`//*`	Selects all the elements in the document, including the root document element.

Table 5-1 *XPath Hierarchy*

expression `//origin[not(. = preceding::origin)]`. The part of this expression containing the instruction `preceding::` is used to search along the sibling axis, selecting all preceding siblings of the type `origin`. The `.` is used to select the value of the current `origin` element. The entire expression translates to: select all the `origin` elements in the document whose values do not equal any of the values of the preceding siblings. Essentially, it selects the set of all `origin` elements whose values are distinct. When applied against the XML in Listing 5-8, it selects the first `origin` element that contains the value of `California` and the first `origin` element that has a value of `Florida`.

The second line is a `for-each` element that loops through all the nodes in the variable `origins`. Note the use of `$` to indicate the value of the variable `origins`. In this example, the variable `origins` contains two `origin` nodes: one containing the value of `Florida` and one containing the value of `California`.

The first line in this loop creates a variable named `sameOrigin`. This variable is assigned all the `apples` elements in the document whose child element `origin` has the same value as the value of the current node. Because this is the first time through the loop, the value of the `origin` element is `California`. The net effect is that the `sameOrigin` variable is assigned all the `apples` elements in the document whose child element `origin` has a value of `California`. This is an excellent way to group elements together based on a particular value.

The next line emits a `<p>` tag followed by the value of the current node, which, in this case is `California`, bracketed by `<b>` and `</b>` tags.

Next, a sum is calculated and emitted, surrounded by `<ul><li>` and `</li></ul>` tags, respectively. The sum is generated by using the `sum()` XPath function. It adds together the values of all the `quantity` child elements in the `sameOrigin` variable. Because the `sameOrigin` variable contains only those `apples` elements whose `origin` value equals `California`, this emits the sum of all the apples originating in California. In this case, the value is `1500`.

The next instruction in the template is an `apply-templates` element. The `select` attribute is assigned the value of the `sameOrigin` variable, which is a list of `apples` elements. The template that matches against `apples` elements is then invoked. Because the `apply-templates` element contains a `sort` element, the list of `apples` elements is passed to this new template sorted by the `price` element in ascending order. This template contains two `if` elements; the same thing could also be achieved using the `choose` element. In any case, the first `if` element tests whether the current element is the last element in the list. The `position()` method returns the numeric position of the current node within the list of nodes passed to the template. The `last()` function essentially returns the number of nodes in this list. The first time through, this is obviously false.

The second `if` element tests whether the current node is the first element in the list by comparing the value returned by the `position()` method to the number 1. Of course, the first time through, this tests true, so the element's body is executed. The instructions contained with the element output the following: `<ul><li>` tags, the string `Min:`, the value of the current node's `price` element using the `current()` method to return the current node, and then, finally, the tags `</li></ul>`.

When this template processes the last node in the list, the first `if` element evaluates to true. At this point, the following output is generated: `<ul><li>` tags, the string `Max:`, the value of the current node's `price` element, again, using the `current()` method to return the current node, and then, finally, the terminating tags `</li></ul>`.

At this point, execution is returned to the calling template. It emits a terminating `</p>` tag, and then loops to the next element in the variable `origins` list of nodes. The second, and last, element in that list is the first `apples` element that was found containing an `origin` element with a value of `Florida`. The previous process is then repeated, generating a list of all the `apples` elements that originated in Florida. The sum of quantities is calculated, equaling `2800` in this case. And then, the minimum and maximum prices are determined for this list in the same manner as the list of `California` nodes. The final output appears as shown previously in Figure 5-3.

Expressions

XPath expressions provide XSLT with the power to select nodes based on particular criteria. For example, it offers a way to select only those nodes that satisfy certain constraints or test true against some comparison. The expressions are evaluated and a list of nodes is returned. If no nodes meet the criteria, then the list is empty.

Comparisons like equality, nonequality, less-than, and greater-than relationships are typical examples of expressions. Referencing other portions of the source tree is also possible. Table 5-1 showed some common examples of selecting specific branches using the `AbsoluteLocationPath` and `AbbreviatedAbsoluteLocationPath` expressions.

Another common expression is the `AbbreviatedStep` defined by two symbols: `.`—which evaluates to the current context node and `..`—which refers to the parent of the context node. For example, this line from the end of the stylesheet in Listing 5-9.

```
L 5-35      <xsl:value-of select="current()/price"/>
```

could be written like this

L 5-36
```
<xsl:value-of select="./price"/>
```

and still yield exactly the same result.

Search Axes

In the example shown in Listing 5-9, some of the XPath expressions used a search axis to assemble the proper list of nodes. A *search axis* is a way of looking at the nodes in an XML tree in one specific direction. For example, the `ancestor` axis narrows the search to contain only the lineage of nodes from the document root down to the parent of the current node. It ignores all siblings, focusing instead on the parent node, the grandparent node, and so on all the way back to the beginning. For example, let's examine what happens when the XML document in Listing 5-11 is processed by the XSLT stylesheet in Listing 5-12.

Listing 5-11 *Ancestor.xml*

L 5-37
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="C:\Ancestor.xsl"?>
<root>
    <streetAddress value="123 Main Street">
        <city value="San Francisco">
            <state value="CA">
                <zipCode value="12345">
                    <company>ACME Corp</company>
                </zipCode>
            </state>
        </city>
    </streetAddress>
</root>
```

Listing 5-12 *Ancestor Stylesheet*

L 5-38
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

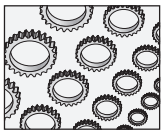
```
    <xsl:output
        method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
    <xsl:template match="/">
    <html>
        <body>
            <xsl:apply-templates select="//company"/>
        </body>
    </html>
    </xsl:template>
    <xsl:template match="company">
        <h1><xsl:value-of select="//company"/></h1>
        <h2>Address:</h2>
        <h3>
            <xsl:apply-templates select="ancestor::*/@value"/>
        </h3>
    </xsl:template>
    <xsl:template match="@value">
        <xsl:value-of select="."/>
        <xsl:choose>
            <xsl:when test="position() &lt; (last() - 1)">, </xsl:when>
            <xsl:when test="position() &lt; last()">&#32;</xsl:when>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

Although the XML document in Listing 5-11 may not represent the ideal way to structure the data, it demonstrates how the `ancestor` axis operates. Applying the stylesheet to the XML in Listing 5-11 generates the following output:

L 5-39

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
    <body>
        <h1>ACME Corp</h1>
        <h2>Address:</h2>
        <h3>123 Main Street, San Francisco, CA 12345</h3>
    </body>
</html>
```

As you can see, the `ancestor` axis generated a list of nodes from the top of the document down to the parent of the current node.



NOTE

An XPath axis operates on the XML nodes in the order they appear in the XML document, ignoring any sorting applied to the nodes. In other words, if a template employs an XPath axis, and the template's list of nodes was previously sorted by an `xsl:sort` statement, the results may be different than expected, as the axis will ignore the sort order of the nodes and recognize their physical ordering instead.

Table 5-2 lists some of the more common axes used in XPath. For a complete list, refer to the XPath specification, section 2.2, rule 6. The reference, "current context node," in the axis descriptions identifies the starting node for the search axis.

Axis	Meaning
`ancestor`	Compiles a list of nodes consisting of the parent node, the grandparent node, and so on up to the document root, ignoring siblings of any kind.
`ancestor-or-self`	This is the same as the `ancestor` axis, but also includes the current context node.
`@ | attribute`	This returns the list of attributes for the context node.
`descendant`	Compiles a list of node children and their children of the original context node, recursively. The sequence of nodes is in the order they appear in the document.
`descendant-or-self`	This is the same as the `descendant` axis, but also includes the current context node.
`following`	Returns the list of nodes that appear after the current context node, excluding ancestors. It won't contain any attribute or namespace nodes (namespace nodes are those nodes prefixed by a namespace).
`following-sibling`	This axis returns the list of sibling nodes that come after the current context node, which have the same parent node as the context node.
`preceding`	Returns the list of nodes that appear before the current context node, excluding ancestors. The sequence of nodes is in reverse document order. It won't contain any attribute or namespace nodes.
`preceding-sibling`	Compiles a list of sibling nodes that come before the current context node and share the same parent node as the context node. They appear in reverse document order.

Table 5-2 *Common XPath Axes*

An axis is specified within an XPath expression by prefixing the node with the axis name and a double colon `::`. The exception to this rule is the attribute axis. When using the abbreviated form of the attribute axis, `@`, the double-colon prefix isn't used.

Functions

Functions play an important role in XPath, providing the capability to convert strings to numbers and numbers to strings, and to calculate new values from existing values. Functions offer XSLT the processing power to transform the source XML content into something new.

For example, the `string()` function evaluates to a string representation of the given value. The function `number()` converts strings into numbers. And `the format-number()` function formats a number given a specific format. This format can further be defined using the `decimal-format` element described earlier in this chapter. The following use of the `format-number()` function

L 5-40
```
<xsl:value-of select="format-number(4567.89, '$#,##0.00')"/>
```

converts the given number into this:

L 5-41
```
$4,567.89
```

Other numeric functions, such as `sum()` and `count()` can calculate totals on a given node set. The `sum()` function adds up the values within the node set, while the `count()` function operates on a given list of nodes and returns the total number of nodes within the set.

More popular functions like `position()` and `last()` return numeric values about the current set of nodes within a particular context. The `position()` function returns the position of the current context node as it relates to the entire set of nodes. The `last()` function returns the numeric position of the last node in the list. Listing 5-9 uses these functions to determine if the current node the template is operating on is either the first node or the last node in the list.

Conclusion

This chapter has shown that XSLT is a rich programming language derived from XML and provides developers with the power to query portions of the source XML and transform it into a desired output form. XSLT has a number of elements defined

in its schema to offer developers with all the tools to generate both new XML documents and dynamic presentation.

XSLT relies on XPath to query and modify portions of a source XML document. XPath uses expressions to select portions of the source XML tree. These expressions can include searches along specific axes. Each axis type prunes the source XML in a particular fashion so the transformation processes only select nodes in the tree. For example, the ancestor axis selects only those nodes that form the direct path from the root node of the document all the way down to the current node's parent—all siblings are ignored. The descendant axis, on the other hand, selects all the current node's children, and their children, and so on, recursively. In addition, XPath provides functions to perform calculations, conversions, and comparisons.

In most of the code examples used in this book, we use XSLT and XPath to generate HTML from XML for consumption by client browsers. This isn't the only purpose of XSLT by far, but it does make transforming pure data content into pure presentation content easy. For a complete reference on XSLT and XPath, read the specifications at the W3C Web site, **http://www.w3.org**.