XSLT and Style Sheets

XSLT and Style Sheets

29-04-2011

Table of Contents

XSLT and Style Sheets

Learning objectives

- Output XML to an XML file
- Create numbered lists
- Use parameters in a stylesheet
- Import multiple stylesheets

In previous chapters, we have introduced the basics of using an XSL stylesheet to convert XML documents into HTML. This chapter will briefly review those concepts and introduce many new ones as well. It is a reference for creating stylesheets.

XML Stylesheets

The eXtensible Stylesheet Language (XSL) provides a means to transform and format the contents of XML document for display. It includes two parts, XSL Transformation (XSLT) for transforming the XML document, and XSLFO (XSL Formatting Objects) for formatting or applying styles to XML documents. The XSL Transformation Language (XSLT) is used to transform XML documents from one form to another, including new XML documents, HTML, XHTML, and text documents. XSL-FO can create PDF documents, as well as other output formats, from XML. With XSLT you can effectively recycle content, redesigning it for use in new documents, or changing it to fit limitless uses. For example, from a single XML source file, you could extract a document ready for print, one for the Web, one for a Unix manual page, and another for an online help system. You can also choose to extract only parts of a document written in a specific language from an XML source that stores text in many languages. The possibilities are endless!

An XSLT stylesheet is an XML document, complete with elements and attributes. It has two kinds of elements, top-level and instruction. Top-level elements fall directly under the `stylesheet`

root element. Instruction elements represent a set of formatting instructions that dictate how the contents of an XML document will be transformed. During the transformation process, XSLT analyzes the XML document, or the source tree, and converts it into a node tree, a hierarchical representation of the entire XML document, also known as the result tree. Each node represents a piece of the XML document, such as an element, attribute or some text content. The XSL stylesheet contains predefined "templates" that contain instructions on what to do with the nodes. XSLT will use the `match` attribute to relate XML element nodes to the templates, and transform them into the result document.

Let's review the stylesheet, city.xsl from chapter 2, and examine it in a little more detail:

Exhibit 1: XML stylesheet for city entity

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

<!--
      Document:  city.xsl
-->

<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="html"/>
   <xsl:template match="/">
      <html>
          <head>
             <title>Cities</title>
          </head>
          <body>
             <h2>Cities</h2>
             <xsl:apply-templates select="cities"/>
          </body>
      </html>
  </xsl:template>
  <xsl:template match="cities">
      <!-- the for-each element can be used to loop through each node in a spe-
cified node set (in this case city) -->
      <xsl:for-each select="city">
         <xsl:text>City: </xsl:text>
           <xsl:value-of select="cityName"/>
           <br/>
         <xsl:text>Population: </xsl:text>
           <xsl:value-of select="cityPop"/>
           <br/>
          <xsl:text>Country: </xsl:text>
          <xsl:value-of select="cityCountry"/>
          <br/>
```

```
            <br/>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

- Since a stylesheet is an XML document, it begins with the XML declaration. This includes the pseudo-attributes `encoding` and `standalone`. They are called pseudo because they are not the same as element attributes. The standalone attribute allows you to directly specify an external DTD
- The `<xsl:stylesheet>` tag declares the start of the stylesheet and identifies the version number and the official W3C namespace. Notice the conventional prefix for the XSLT namespace, xsl. Once a prefix is declared, it must be used for all the elements.
- The <xsl:output> tag is an optional element that determines how to output the result tree.
- The `<xsl:template>` element defines the start of a template and contains rules to apply when a specified node is matched. The `match` attribute is used to `associate`(match) the template with an XMLelement, in this case the root (/), or whole branch, of the XML source document.
- If no output method has been specified, the output would default to HTML in this case since the root element is the `<html>` start tag
- The `apply-templates` element is an empty element since it has no character content. It applies a template rule to the current element or the element's child nodes. The `select` attribute contains a location path telling it which element's content to process.
- The instruction element `value-of` extracts the string value of the child of the selected node, in this case, the text node child of `cityName`

The `template` element defines the rules that implement a change. This can be any number of things, including a simple plain-text conversion, the addition or removal of XML elements, or simply a conversion to HTML, when the pattern is matched. The pattern, defined in the element's `match` attribute, contains an abbreviated XPath location path. This is basically the name of the root element in the doc, in our case, "tourGuide."

When transforming an XML document into HTML, the processor expects that elements in the stylesheet be well-formed, just as with XML. This means that all elements must have an end tag. For example, it is not unusual to see the `<p>` tag alone. The XSLT processor requires that an element with a start-tag must close with an end tag. With the `<br>` element, this means either using `<br></br>` or `<br />`. As mentioned in Chapter 3, the `br` element is an empty element. That means it carries no content between tags, but it may have attributes. Although no end tags are output

for the HTML output, they still must have end-tags in the stylesheet. For instance, in the stylesheet, you will list: `<img src="picture.jpg"></img>` or as an empty element `<img src="picture.jpg" />` . The HTML output will drop the end-tag so it looks like this: `<img src="picture.jpg">` On a side note, the processor will recognize html tags no matter what case they are in - BODY, body, Body are all interpreted the same.

Output

XSLT can be used to transform an XML source into many different types of documents. XHTML is also XML, if it is well formed, so it could also be used as the source or the result. However, transforming plain HTML into XML won't work unless it is first turned into XHTML so that it conforms to the XML 1.0 recommendation. Here is a list of all the possible type-to-type transformations performed by XSLT:

Exhibit 2: Type-To-Type Transformations

	XML	XHTML	HTML	text
XML	X	X	X	X
XHTML	X	X	X	X
HTML				
text				

The `output` element in the stylesheet determines how to output the result tree. This element is optional, but it allows you to have more control over the output. If you do not include it, the output method will default to XML, or HTML if the first element in the result tree is the `<html>` element. Exhibit 3 lists attributes.

Exhibit 3: Element output attributes (from Wiley: XSL Essentials by Michael Fitzgerald)

Attribute	Description
cdata-section-elements	Specifies a list of whitespace-separated element names that will contain CDATA sections in the result tree. A CDATA escapes characters that are normally interpreted as markup, such as a < or an &.

doctype-public	Places a public identifier in a document type declaration in a result tree.
doctype-system	Places a public identifier in a document type declaration in a result tree.
encoding	Sets the preferred encoding type, such as UTF-8, ISO-8859, etc. These values are not case sensitive.
indent	Indicates that the XSLT processor may indent content in the result tree. Possible values are `"yes"` or `"no"`. The default is no when `method="xml"`.
media-type	Sets the media type (MIME type) for the content of the result tree.
method	Specifies the type of output. Legal values are `xml, html, text,` or another qualified name.
omit-xml-declaration	Tells the XSLT processor to include or not include an XML declaration
standalone	Tells the XSLT processor to include a pseudo-attribute in the XML declaration (if not omitted) with a value of either `"yes"` or `"no"`. This indicates whether the document depends on external markup declarations, such as those in an external DTD.
version	Sets the version number for the output method such as the version of XML used for output (default is `1.0`)

XML to XML

Since we have had a lot of practice transforming an XML document to HTML, we are going to transform city.xml, used in chapter 2, into another XML file, using host.xsd as the schema.

Exhibit 4: XML document for city entity

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

<!--
    Document:  city.xml
-->
```

```
<cities xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='host.xsd'>
<city>
    <cityID>c1</cityID>
    <cityName>Atlanta</cityName>
    <cityCountry>USA</cityCountry>
    <cityPop>4000000</cityPop>
    <cityHostYr>1996</cityHostYr>

</city>

<city>
    <cityID>c2</cityID>
    <cityName>Sydney</cityName>
    <cityCountry>Australia</cityCountry>
    <cityPop>4000000</cityPop>
    <cityHostYr>2000</cityHostYr>
    <cityPreviousHost>c1</cityPreviousHost >
</city>

<city>
    <cityID>c3</cityID>
    <cityName>Athens</cityName>
    <cityCountry>Greece</cityCountry>
    <cityPop>3500000</cityPop>
    <cityHostYr>2004</cityHostYr>
    <cityPreviousHost>c2</cityPreviousHost >
</city>

</cities>
```

Exhibit 5: XSL document for city entity that list cities by City ID

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <xsl:for-each select="//city[count(cityPreviousHost) = 0]">
            <br/><xsl:text>City Name: </xsl:text><xsl:value-of select="city-
Name"/><br/>
            <xsl:text>           Rank: </xsl:text><xsl:value-of select="city-
ID"/><br/>
            <xsl:call-template name="output">
                <xsl:with-param name="context" select="."/>
            </xsl:call-template>
        </xsl:for-each>
```

```
    </xsl:template>
    <xsl:template name="output">
        <xsl:param name="context" select="."/>
        <xsl:for-each select="//city[cityPreviousHost = $context/cityID]">
            <br/><xsl:text>City Name: </xsl:text> <xsl:value-of select="city-
Name"/><br/>
            <xsl:text>            Rank: </xsl:text><xsl:value-of select="city-
ID"/><br/>
            <xsl:call-template name="output">
                <xsl:with-param name="context" select="."/>
            </xsl:call-template>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Exhibit 6: XML schema for host city entity

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qual-
ified"
attributeFormDefault="unqualified">

<xsd:element name="cities">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="city" type="cityType" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="cityType">
  <xsd:sequence>
    <xsd:element name="cityID" type="xsd:ID"/>
     <xsd:element name="cityName" type="xsd:string"/>
     <xsd:element name="cityCountry" type="xsd:string"/>
     <xsd:element name="cityPop" type="xsd:integer"/>
     <xsd:element name="cityHostYr" type="xsd:integer"/>
     <xsd:element name="cityPreviousHost" type="xsd:IDREF" minOccurs="0" maxOc-
curs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

Exhibit 7: XML stylesheet for city entity

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

<!--
    Document:  city2.xsl
-->

   <xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" encoding="utf-8" indent="yes" />
   <xsl:attribute-set name="date">
      <xsl:attribute name="year">2004</xsl:attribute>
      <xsl:attribute name="month">03</xsl:attribute>
      <xsl:attribute name="day">19</xsl:attribute>
   </xsl:attribute-set>
       <xsl:template match="tourGuide">
       <xsl:processing-instruction name="xsl-stylesheet">  href="style.css"
type="text/css"<br />
           </xsl:processing-instruction>
       <xsl:comment>This is a list of the cities we are visiting this
week</xsl:comment>
          <xsl:for-each select="city">

<!-- element name creates a new element where the value of the attribute name
sets name of
the new element.  Multiple attribute sets can be used in the same element -->

<!-- use-attribute-sets attribute adds all the attributes declared in attribute-
set from above -->

              <xsl:element name="cityList"  use-attribute-sets="date">
                  <xsl:element name="city">
                        <xsl:attribute name="country">
                     <xsl:apply-templates select="country"/>  </xsl:attribute>

                        <xsl:apply-templates select="cityName"/>
                  </xsl:element>
                  <xsl:element name="details">Will write up a one page report
 of the trip</xsl:element>
              </xsl:element>
         </xsl:for-each>
      </xsl:template>
   </xsl:stylesheet>
```

- Although the `output method` is set to "xml", since there is no `<html>` element as the root of the result tree, it would default to XML output.

- `attribute-set` is a top-level element that creates a group of attributes by the name of "date." This attribute set can be reused throughout the stylesheet. The element `attribute-set` also has the attribute use-attribute-sets allowing you to chain together several sets of attributes.
- The `processing-instruction`produces the XML stylesheet processing instructions.
- The element `comment` creates a comment in the result tree
- The `attribute` element allows you to add an attribute to an element that is created in the result tree.

The stylesheet produces this result tree:

Exhibit 8: XML result tree for city entity

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>

<!--
    Document:  city2.xsl
-->

  <?xsl-stylesheet href="style.css" type="text/css"?>
    <!--This is a list of the cities we are visiting this week-->
    <cityList year="2004" month="03" day="19">
      <city country="Belize">Belmopan</city>
      <details>Will write up a one page report of the trip</details>
    </cityList>
    <cityList year="2004" month="03" day="19">
      <city country="Malaysia">Kuala Lumpur</city>
      <details>Will write up a one page report of the trip</details>
    </cityList>
  </stylesheet>
```

The processor automatically inserts the XML declaration at the top of the result tree. The processing instruction, or PI, is an instruction intended for use by a processing application. In this case, the href points to a local stylesheet that will be applied to the XML document when it is processed. We used `<xsl:element>` to create new content in the result tree and added attributes to it.

There are two other instruction elements for inserting nodes into a result tree. These are `copy` and `copy-of`. Unlike `apply-templates`, which only copies content of the child node (like the child text node), these elements copy everything. The following code shows how the copy element can be used to copy the city element in city.xml:

Exhibit 9: Copy element

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

The result looks like this:

Exhibit 10: Copy element result

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

The output isn't very interesting, because copy does not pick up the child nodes, only the current node. In our example, it picks up the two city nodes that are in the city.xml file. The copy element has an optional attribute, use-attribute-sets, which allows you to add attributes to the element. However, it will leave behind any other attributes, except the namespace, if it is present. Here is the result if a namespace is declared in the source document, in this case, the default namespace:

Exhibit 11: Namespace result

```
<?xml version="1.0" encoding="utf-8">
<city xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<city xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

If you want to copy more from the source file than just one node, the `copy-of` element includes the current node, and any attribute nodes that are associated with it. This includes any nodes that might be laying around, such as namespace nodes, text nodes, and child element nodes. When we apply the `copy-of` element to city.xml, the result is almost an exact replica of city.xml! You can also copy comments and processing instructions using `<xsl:copy-of select="comment()"/>` and `<xsl:copy-of select="processing-instruction(`*name*`)"/>` where *name* is the value of the name attribute in the processing instruction you wish to retrieve.

Why would this be useful, you ask? Sometimes you want to just grab nodes and go! For example, if you want to place a copy of city.xml into a SOAP envelope, you can easily do it using `copy-of`. If you don't already know, Simple Object Access Protocol, or SOAP, is a protocol for packaging XML documents for exchange. This is really useful in a B2B environment because it provides a standard way to package XML messages. You can read more about SOAP at www.w3.org/tr/soap.

Use an XML editor to create the above XML Stylesheets, and experiment with the `copy` and `copy-of` elements.

Templates

Since templates define the rules for changing nodes, it would make sense to reuse them, either in the same stylesheet or in other stylesheets. This can be accomplished by naming a template, and then calling it with a `call-template` element. Named templates from other stylesheets can also be included. You can quickly see how this is useful in practical applications. Here is an example using named templates:

Exhibit 110: Named templates

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" />
    <xsl:template match=" /">
        <xsl:call-template name="getCity" />
    </xsl:template>

    <xsl:template name="getCity">
        <xsl:copy-of select="city" />
    </xsl:template>
</xsl:stylesheet>
```

Templates also have a mode attribute. This allows you to process a node more than once, producing a different result each time, depending on the template. Let's create a stylesheet to practice modes.

Exhibit 12: XML template modes

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

<!--
    Document:  cityModes.xsl
-->

<xsl:stylesheet version="1.0"
```

```
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" />
     <xsl:template match="tourGuide">
        <html>
           <head>
              <title>City - Using Modes</title>
           </head>
           <body>
              <xsl:for-each select="city">
                 <xsl:apply-templates select="cityName" mode="title" />
                 <xsl:apply-templates select="cityName" mode="url" />
                 <br />
              </xsl:for-each>
           </body>
        </html>
     </xsl:template>

     <xsl:template match="cityName" mode="title">
        <h2><xsl:value-of select="current()"/></h2>
     </xsl:template>
      <xsl:template match="cityName" mode="message">
         <p>Come visit <b><xsl:value-of select="current()" /></b>!</p>
     </xsl:template>
</xsl:stylesheet>
```

- `apply-templates select="cityName" mode="title"` tells the processor to look for a template that has the same mode attribute value
- `value-of select="current()"` returns the current node which is converted to a string with `value-of`. Using `select="."` will also return the current node.

The result isn't very flattering since we didn't do much with the file, but it gets the point across.

Exhibit 13: Result from above stylesheet

```
<h2>Belmopan</h2>
Come visit <b>Belmopan</b>!

<h2>Kuala Lumpur</h2>
Come visit <b>Kuala Lumpur</b>!
```

By default, XSLT processors have built-in template rules. If you apply a stylesheet without any matching rules, and it fails to match a pattern, the default rules are automatically applied. The default rules output the content of all the elements.

Sorting

Writing “well formed” code XML is vital. At times, however, simply displaying information (the most elementary level of data management) is not all that is necessary to properly identify a project. As information technology specialists, it is necessary to fully understand that order is vital for interpretation. Order can be attained by putting data in a format that is quickly readable. Such information then becomes quickly usable. Using a comparative model or simply looking for a specific name or item becomes very easy. Finding a specific musical artist, title, or musical type becomes very easy. As an Information Specialist, you must fully be aware that it often becomes necessary to sort information. The basis of sorting in XMLT is the xsl:sort command. The xsl:sort element exemplifies a sort key component. A sort key component identifies how a sort key value is to be identified for each item in the order of information being sorted. A Sort Key Value is defined as “the value computed for an item by using the Nth sort key component” The significance of a sort key component is realized either by its select attribute, or by the contained sequence constructor. A Sequence Constructor is defined as a “sequence of zero or more sibling nodes in the stylesheet that can be evaluated to return a sequence of nodes and atomic values”. There are instances when neither is present. Under these circumstances, the default is select=".", which has the effect of sorting on the actual value of the item if it is an atomic value, or on the typed-value of the item if it is a node. If a select attribute is present, its value must be an Xpath expression.

The following is how the <xsl:sort> element is used to sort the output.

Sort Information is held as Follows: Sorting output in XML is quite easy and is done by adding the <xsl:sort> element after the <xsl:for-each> element in the XSL file.

Exhibit 14: Stylesheet with sort function

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
   <body>
    <h2>TourGuide Example</h2>
```

```
       <xsl:apply-templates select="cities"/>
   </body>
  </html>
    </xsl:template>
    <xsl:template match="cities">
       <xsl:for-each select="city">
         <xsl:sort select="cityName"/>
         <xsl:value-of select="cityName"/>
         <xsl:value-of select="cityCountry"/>
      </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

This example will sort the file alphabetically by artist name. Note: The select attribute indicates what XML element to sort on. Information can be SELECTED and SORTED by “title” or “artist”. These are categories that the XML document will display within the body of the file.

We have used the `sort` function to sort the results of an `if` statement before. The sort element has many other uses as well. Essentially, it instructs the processor to sort nodes based on certain criteria, which is known as the sort key. It defaults to sorting the elements in ascending order. Here is a short list of the different attributes that sort takes:

Exhibit 15: Sort attributes

Attribute	**Description**
`select`	Specifies the node on which to process
`order`	Specifies the sort order: "ascending" or "descending"
`case-order`	Determines whether text in uppercase is sorted before lowercase: "upper-first" or "lower-first"
`data-type`	By default sorts on text data: "text", "number", or QName(qualified name)
`lang`	Indicates the language in use since some languages use different alphabets. "en", "de", "fr", etc. If no value is specified, the language is determined from the system environment.

The sort element can be used in either the `apply-templates` or the `for-each` elements. It can also be used multiple times within a template, or in several templates, to create sub-ordering levels.

Numbering

The `number` instruction element allows you to insert numbers into your results. Combined with a sort element, you can easily create numbered lists. When this simple stylesheet, hotelNumbering.xsl, is applied to city_hotel.xml, we get the result listed below:

Exhibit 16: Sorting and numbering lists

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!--
     Document:  hotelNumbering.xsl
 -->

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

<xsl:output method="text" omit-xml-declaration="yes"/>
    <xsl:template match="/">
         <xsl:apply-templates select="tourGuide/city/hotel">
            <xsl:sort/>
         </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="hotel">
         <xsl:number value="position()" format="&#xa; 0. "/>
         <xsl:value-of select="hotelName"/>
   </xsl:template>
</xsl:stylesheet>
```

Exhibit 17: Result hotelNumbering.xsl

```
 1. Bull Frog Inn
 2. Mandarin Oriental Kuala Lumpur
 3. Pan Pacific Kuala Lumpur
 4. Pook's Hill Lodge
```

The expression in `value` is evaluated and the value for `position()` is based on the sorted node list. To improve the looks we are adding the format attribute with a linefeed character reference (
), a zero digit to indicate that the number will be a zero digit to indicate that the number will be an integer type, and a period and space to make it look nicer. The format list can be based on the following sequences:

Exhibit 17: Numbering formats

```
format=" A. "     -    Uppercase letters
format=" a. "     -    Lowercase letters
format=" I. "     -    Uppercase Roman numerals
format=" i. "     -    Lowercase Roman numerals
format=" 000. "   -    Numeral prefix
format=" 1- "     -    Integer prefix/ hyphen prefix
```

To specify different levels of numbering, such as sections and subsections of the source document, the `level` attribute is used, which tells the processor the levels of the source tree that should be considered. By default, it is set to `single`, as seen in the example above. It also can take values of `multiple` and `any`. The `count` attribute is a pattern that tells the processor which nodes to count (for numbering purposes). If it is not specified, it defaults to a pattern matching the same node type as the current node. The `from` attribute can also be used to specify the node where the counting should start.

When level is set to `single`, the processor searches for nodes that match the value of `count`, and if it is not present, it matches the current node. When it finds the match, it creates a node-list and counts all the matching nodes of that type. If the `from` attribute is listed, it tells the processor where to start counting from, rather than counting all nodes.

When the level is `multiple`, it doesn't just count a list of one node type, it creates a list of all the nodes that are ancestors of the current node, in the actual order from the source document. After this list is created, it selects all the nodes that match the nodes represented in count. It then maps the number of preceding siblings for each node that matches count. In effect, `multiple` remembers all the nodes separately. This is where `any` is different. It will number all the elements sequentially, instead of counting them in multiple levels. As with the other two values, you can use the `from` attribute to tell the processor where to start counting from, which in effect will separate it into levels.

This is a modification of the example above using the `level="multiple"`:

Exhibit 18: Sorting and numbering lists

```
<!--
    Document:  hotelNumbering2.xsl
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text" omit-xml-declaration="yes"/>
     <xsl:template match="/">
          <xsl:apply-templates select="tourGuide//hotelName"/>
     </xsl:template>

       <xsl:template match="hotel">
            <xsl:number level="multiple"
                  count="city|hotel" format="&#xa; 1.1 "/>
            <xsl:apply-templates />
     </xsl:template>
</xsl:stylesheet>
```

Exhibit 19: Result – hotelNumbering2.xsl

```
1.1 Bull Frog Inn
1.2 Pook's Hill Lodge
2.1 Pan Pacific Kuala Lumpur
2.2 Mandarin Oriental Kuala Lumpur
```

The first template matches the root node and then selects all `hotel` nodes that have `country` as an ancestor, creating a node-list. The next template recursively processes the `amenityName` element, and gives it a number for each instance of `amenityName` based on the number of elements in the attribute. This is figured out by counting the number of preceding siblings, plus 1.

Formatting

Formatting numbers is a simple process so this section will be a brief overview of what can be done. Placed within the XML stylesheet, functions can be used to manipulate data during the transformation. In order to make numbers a little easier to read, we need to be able to separate the digits into groups, or add commas or decimals. To do this we use the `format-number()` function. The purpose of this function is to convert a numeric value into a string using specified patterns that control the number of leading zeroes, separator between thousands, etc. The basic syntax of this function is as follows: `format-number (number, pattern)`

- `numbers`
- `pattern` is a string that lays out the general representation of a number. Each character in the string represents either a digit from number or some special punctuation such as a comma or minus sign.

The following are the characters and their meanings used to represent the number format when using the format-number function within a stylesheet:

Exhibit 20: Format-number function

Symbol	Meaning
0	A digit.
#	A digit, zero shows as absent.
. (period)	Placeholder for decimal separator.
,	Placeholder for grouping separator.
;	Separate formats.
-	Default prefix for negative.
%	Multiply by 100 and show as a percentage.
X	Any other characters can be used in the prefix or suffix.
'	Used to quote special characters in a prefix or suffix.

Conditional Processing

There are times when it is necessary to display output based on a condition. There are two instruction elements that let you conditionally determine which template will be used based on certain tests. These are the `if` and `choose` elements.

The test condition for an `if` statement must be contained within the `test` attribute of the `<xsl:if>` element. Expressions that are testing greater than and less than operators must represent them by ">" and "<" respectively in order for the appropriate transformation to take place. The `not()` function from XPath is a Boolean function and evaluates to true if its argument is false, and vice versa. The `and` and `or` conditions can be used to combine multiple tests, but an `if` statement can, at most, test only one expression. It can also only instantiate the use of one template.

The `when` element, is similar to the `else` statement in Java. By using the `when` element, the `choose` element can offer a many alternative expressions. A choose element must contain at least one when statement, but it can have as many as it needs. The choose element can also contain one instance of the otherwise element, which works like the final else in a Java program. It contains the template if none of the other expressions are true.

The `for-each` element is another conditional processing element. We have used it in previous chapter exercises, so this will be a quick review. The `for-each` element is an instruction element, which means it must be children of template elements. `for-each` evaluates to a node-set, based on the value of the select attribute, or expression, and processes through each node in document order, or sorted order.

Parameters and Variables

XSLT offers two similar elements, `variable` and `param`. Both have a required `name` attribute, and an optional `select` attribute, and you declare them like this:

Exhibit 21: Variable and parameter declaration

```
<xsl:variable name="var1" select=""/>
<xsl:param name="par1" select=""/>
```

The above declarations have bound to an empty string, which is the same effect as if you had left off the select attribute. With parameters, this value is considered only a default, or initial value to be changed either from the command line, or from another template using the `with-param` element. However, with the variable, as a general rule, the value is set and can't be changed dynamically except under special circumstances. When making declarations, remember that variables can be declared anywhere within a template, but a parameter must be declared at the beginning of the template.

Both elements can also have global and local scope, depending on where they are defined. If they are defined at the top-level under the <stylesheet> elements, they are global in scope and can be used anywhere in the stylesheet. If they are defined in a template, they are local and can only be used in that template. Variables and parameters declared in templates are visible only to the template they are declared in, and to templates underneath them. They have a cascading effect: they can spill down from the top-level into a template, down into a template within that one, etc, but they cannot go back up!

We are going to hard-code a value for the parameter in it's declaration element using the `select` attribute.

Exhibit 22: HTML results

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

<!--
    Document:  countryParam.xsl
-->

  <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
    <xsl:param name="country" select="'Belize'"/>
    <xsl:param name="code" />
  <xsl:template match="/">
          <xsl:apply-templates select="country-codes" />
   </xsl:template>
       <xsl:template match="country-codes">
         <xsl:apply-templates select="code" />
    </xsl:template>
    <xsl:template match="code">
         <xsl:choose>
            <xsl:when test="countryName[. = $country]">
                      The country code for
                      <xsl:value-of select="countryName"/> is
                      <xsl:value-of select="countryCode"/>.
            </xsl:when>
            <xsl:when test="countryCode[. = $code]">
                      The country for the code
                      <xsl:value-of select="countryCode"/> is
                      <xsl:value-of select="countryName"/>.
            </xsl:when>
            <xsl:otherwise>
                      Sorry. No matching country name or country code.
            </xsl:otherwise>
          </xsl:choose>
        </xsl:template>
      </xsl:stylesheet>
```

The value that you pass in does not have to be enclosed in quotes, unless you are passing a value with more than one word. For example, we could have passed either country="United States" or country=Belize without getting an error.

The value of a variable can also be used to set an attribute value. Here is an example setting the countryName element with an attribute of countryCode equal to the value in the`$code` variable:

Exhibit 23: Attribute of countryCode

```
<countryName countryCode="{$code}"></countryName>
```

This is known as an attribute value template. Notice the use of braces around the parameter. This tells the processor to evaluate the content as an expression, which then converts the result to a string in the result tree. There are attributes which cannot be set with an attribute value template:

- Attributes that contain patterns (such as `select` in `apply-templates`)
- Attributes of top-level elements
- Attributes that refer to named objects (such as the `name` attribute of `template`)

Parameters, though not variables, can be passed between templates using the `with-param` element. This element has two attributes, `name`, which is required, and `select`, which is optional. This next example uses with-param as a child of the `call-template` element, although it can also be used as a child of `apply-templates`.

Exhibit 24: XSL With-Param

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

<!--
    Document:  withParam.xsl
-->

    <xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output method="text"/>
        <xsl:template match="/">
            <xsl:apply-templates select="tourGuide/city"/>
        </xsl:template>
        <xsl:template match="city">
            <xsl:call-template name="countHotels">
                <xsl:with-param name="num" select="count(hotel)"/>
            </xsl:call-template>
        </xsl:template>
        <xsl:template name="countHotels">
            <xsl:param name="num" select="'''" />
            <xsl:text>City Name: </xsl:text>
            <xsl:value-of select="cityName" />
            <xsl:text>&#xa;</xsl:text>
            <xsl:text>Number of hotels: </xsl:text>
            <xsl:value-of select="$num" />
            <xsl:text>&#xa;&#xa;</xsl:text>
```

```
        </xsl:template>
    </xsl:stylesheet>
```

- `<xsl:template match="city">`Here we match the `city` nodes that were returned in the `apply-templates` node set.
- `call-template`, as discussed earlier, calls the template named`countHotels`
- The element `with-param` tells the called template to use the parameter named `num`, and the select statement sets the expression that will be evaluated.
- Notice the declaration for the parameter is in the first line of the template. It instantiates `num` to an empty string, because the value will be replaced by the value of the expression in the `with-param` element's `select` attribute.
- `
` outputs a line feed in the result tree to make the output look nicer.

Exhibit 25: Text results – withParam.xsl

```
City Name: Belmopan
     Number of hotels: 2

City Name: Kuala Lumpur
     Number of hotels: 2
```

The Muenchian Method

The Muenchian Method is a method developed by Steve Muench for performing functions using keys. Keys work by assigning a key value to a node and giving you access to that node through the key value. If there are lots of nodes that have the same key value, then all those nodes are retrieved when you use that key value. Effectively this means that if you want to group a set of nodes according to a particular property of the node, then you can use keys to group them together. One of the more common uses for the Muenchian method is grouping items and counting the number of occurrences in that group, such as number of occurrences of a city

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="html"/>
    <xsl:key name="Count" match="*/city" use="cityName" />
    <xsl:template match="cities">
```

```
        <xsl:for-each
          select="//city[generate-id()=generate-id(key('Count', cityName)[1])]">

            <br/><xsl:text>City Name:</xsl:text><xsl:value-of select="city-
Name"/><br/>
            <xsl:text>Number of Occurences:</xsl:text>
            <xsl:value-of select="count(key('Count', cityName))"/>
            <br/>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Text Results – muenchianMethod.xsl

```
     City Name: Atlanta
          Number of Occurrences: 1
     City Name: Athens
          Number of Occurrences: 1
     City Name: Sydney
          Number of Occurrences: 1
```

Datatypes

There are five different datatypes in XSLT: Node-set, String, Number, Boolean, and Result tree fragment. Variables and parameters can be bound to each of these, but the last type is specific to them.

Node-sets are returned everywhere in XSLT. We've seen them returned from `apply-templates` and `for-each` elements, and variables. Now we will see how a variable can be bound to a node-set. Examine the following code:

Exhibit 26: Variable bound to a node-set

```
<xsl:variable name="cityNode"  select="city" />
...
<xsl:template match="/">
     <xsl:apply-templates select="$cityNode/cityName" />
</xsl:template>
```

Here, we are setting the value of the variable `$cityNode` to the node-set `city` from the source tree. The `cityName` element is a child of city, so the output generated by apply-templates is the

text node of `cityName`. Remember, you can use variable references in expressions but not patterns. This means we cannot use the reference `$cityNode` as the value of a `match` attribute.

String types are useful if you are interested only in the text of nodes, rather than in the whole node-set. String types use XPath functions, most notably, `string()`. This is just a simple example:

Exhibit 27: String types

```
<xsl:variable name="cityName" select="string('Belmopan')" />
```

This is in fact, a longer way of saying:

Exhibit 28: Shorter version of above

```
<xsl:variable name="cityName" select="' Belmopan'" />
```

It is also possible to declare a variable that has a number value. You do this by using the XPath function `number()`.

Exhibit 29: Declaration of variable with number value

```
<xsl:variable name="population" select="number(11100)" />
```

You can use numeric operators such as + - * / to perform mathematic operations on numbers, as well as some built in XPath functions such as `sum()` and `count()`.

The Boolean type has only two possible values, true or false. As an example, we are going to use a Boolean variable to test to see if a parameter has been passed into the stylesheet.

Exhibit 30: Boolean variable to test

```
<xsl:param name="isOk" select="''" />
<xsl:template match="city" />
     <xsl:choose>
          <xsl:when test="boolean($isOk)">
               …logic here…
          </xsl:when>
          <xsl:otherwise>
               Error: must use parameter isOk with any value to apply template
          </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

We start with an empty-string declaration for the parameter `isOk`. In the `test` attribute of `when`, the `boolean()` function tests the value of `isOk`. If the value is an empty string, as we defined by default, `boolean()` evaluates to `false()`, and the template is not instantiated. If it does have a value, and it can be any value at all, `boolean()` evaluates to `true()`.

The final datatype is the result tree fragment. Essentially it is a chunk of text (a string) that can contain markup. Let's look at an example before we dive into the details:

Exhibit 31: Result tree fragment datatype

```
<xsl:variable name="fragment">
<description>Belmopan is the capital of Belize</description>
</xsl:variable>
```

Notice we didn't use the select attribute to define the variable. We aren't selecting a node and getting its value, rather we are creating arbitrary text. Instead, we declared it as the content of the element. The text in between the opening and closing variable tags is the actual fragment of the result tree. In general, if you use the select attribute as we did earlier, and don't specify content when declaring variables, the elements are empty elements. If you don't use select and you do specify content, the content is a result tree. You can perform operations on it as if it were a string, but unlike a node set, you can't use operators such as / or // to get to the nodes. The way you retrieve the content from the variable and get it into the result tree is by using the copy-of element. Let's see how we would do this:

Exhibit 32: Retrieve and place into result tree

```
<xsl:template match="city"
     <xsl:copy-of select="cityName" />
     <xsl:copy-of select="$fragment" />
</xsl:template>
```

The result tree would now contain two elements: a copy of the city element and the added element, description.

EXSLT

EXSLT is a set of community developed extensions to XSLT. The modules include facilities to handle dates and times, math, and strings.

Multiple Stylesheets

In previous chapters, we have imported and used multiple XML and schema documents. It is also possible to use multiple stylesheets using the `import` and `include` elements, which should be familiar. It is also possible to process multiple XML documents at a time, in one stylesheet, by using the XSLT function `document()`.

Including an external stylesheet is very similar to what we have done in earlier chapters with schemas. The `include` element only has one attribute, which is `href`. It is required and always contains a URI (Uniform Resource Identifier) reference to the location of the file, which can be local (in the same local directory) or remote. You can include as many stylesheets as you need, as long as they are at the top level. They can be scattered all over the stylesheet if you want, as long as they are children of the `<stylesheet>` element. When the processor encounters an instance of include, it replaces the instance with all the elements from the included document, including template rules and top-level elements, but not the root `<stylesheet>` element. All the items just become part of the stylesheet tree itself, and the processor treats them all the same. Here are declarations for including a local and remote stylesheet:

Exhibit 33: Declarations for local and remote stylesheet

```
<xsl:include href="city.xsl" />
<xsl:include href="http://www.somelocation.com/city.xsl"/>
```

Since `include` returns all the elements in the included stylesheet, you need to make sure that the stylesheet you are including does not include your own stylesheet. For example, city.xsl cannot include city_hotel.xsl, if city_hotel.xsl has an include element which includes city.xsl. When including multiple files, you need to make sure that you are not including another stylesheet multiple times. If city_hotel.xsl includes amenity.xsl, and country.xsl includes amenity.xsl, and city.xsl includes both city_hotel.xsl and country.xsl, it has indirectly included amenity.xsl twice. This could cause template rule duplication and errors. These are some confusing rules, but they are easy to avoid if you carefully examine the stylesheets before they are included.

The difference between importing stylesheets and including them is that the template rules imported each have a different import precedence, while included stylesheet templates are merged into one tree and processed normally. Imported templates form an import tree, complete with the root `<stylesheet>` element so the processor can track the order in which they were imported. Just like include, import has one attribute, href, which is required and should contain the URI reference for the document. It is also a top-level element and can be used as many times as need. However, it

must be the immediate child for the <stylesheet> element, otherwise there will be errors. This code demonstrates importing a local stylesheet:

Exhibit 34: Importing local stylesheet

```
<xsl:import href="city.xsl" />
```

The order of the import elements dictates the precedence that matching templates will have over one another. Templates that are imported last have higher priority than those that are imported first. However, the template element also has a priority attribute that can affect its priority. The higher the number in the priority attribute, the higher the precedence. Import priority only comes into effect when templates collide, otherwise importing stylesheets is not that much different from including them. Another way to handle colliding templates is to use the apply-imports element. If a template in the imported document collides with a template in the importing document, apply-templates will override the rule and cause the imported template to be invoked.

The document() function allows you to process additional XML documents and their nodes. The function is called from any attribute that uses an expression, such as the select attribute. For example:

Exhibit 35: Document() function

```
<xsl:template match="hotel">
     <xsl:element name="amenityList">
          <xsl:copy-of select="document('amenity.xml')" />
     </xsl:element>
</xsl:template>
```

When applied to an xml document that only contains an empty hotel element, such as <hotel></hotel>, the result tree will add a new element called amenityList, and place all the content from amenity.xml (except the XML declaration) in it. The document function can take many other parameters such as a remote URI, and a node-set, just to name a few. For more information on using document(), visit http://www.w3.org/TR/xslt#document

XSL-FO

XSL-FO stands for Extensible Stylesheet Language Formatting Objects and is a language for formatting XML data. When it was created, XSL was originally split into two parts, XSL and XSL-

FO. Both parts are now formally named XSL. XSL-FO documents define a number of rectangular areas for displaying output. XSL-FO is used for the formatting of XML data for output to screen, paper or other media, such as PDF format. For more information, visit http://www.w3schools.com/xslfo/default.asp

Summary

XML stylesheets can output XML, text, HTML or XHTML. When an XSL processor transforms an XML document, it converts it to a result tree of nodes, each of which can be manipulated, extracted, created, or set aside, depending on the rules contained in the stylesheet. The root element of a stylesheet is the `<stylesheet>` element. Stylesheets contain top-level and instruction elements. Templates use XPath locations to match a pattern of nodes in the source tree, and then apply defined rules to the nodes when it finds a match. Templates can be named, have a mode, or a priority. Node sets from the source tree can be sorted or formatted. XSLT uses for-each and if elements for conditional processing. XSLT also supports the use of variables and parameters. There are five basic datatypes: a node-set, a string, a number, a Boolean, and a result tree fragment. A stylesheet can also `include` or `import` additional stylesheets or even additional XML documents. XSL-FO is used for formatting data into rectangular objects.

Reference Section

Exhibit 36: XSL Elements (from http://www.w3schools.com/xsl/xsl_w3celementref.asp and http://www.w3.org/TR/xslt#element-syntax-summary)

Element	Description	Category
apply-imports	Applies a template rule from an imported stylesheet	instruction
apply-templates	Applies a template rule to the current element or to the current element's child nodes	instruction
attribute	Adds an attribute	instruction

attribute-set	Defines a named set of attributes	top-level-element
call-template	Calls a named template	instruction
choose	Used in conjunction with <when> and <otherwise> to express multiple conditional tests	instruction
comment	Creates a comment node in the result tree	instruction
copy	Creates a copy of the current node (without child nodes and attributes)	instruction
copy-of	Creates a copy of the current node (with child nodes and attributes)	instruction
decimal-format	Defines the characters and symbols to be used when converting numbers into strings, with the format-number() function	top-level-element
element	Creates an element node in the output document	instruction
fallback	Specifies an alternate code to run if the processor does not support an XSLT element	instruction
for-each	Loops through each node in a specified node set	instruction
if	Contains a template that will be applied only if a specified condition is true	instruction
import	Imports the contents of one stylesheet into another. **Note:** An imported stylesheet has lower precedence than the importing stylesheet	top-level-element

include	Includes the contents of one stylesheet into another. **Note:** An included stylesheet has the same precedence as the including stylesheet	top-level-element
key	Declares a named key that can be used in the stylesheet with the key() function	top-level-element
message	Writes a message to the output (used to report errors)	instruction
namespace-alias	Replaces a namespace in the stylesheet to a different namespace in the output	top-level-element
number	Determines the integer position of the current node and formats a number	instruction
otherwise	Specifies a default action for the <choose> element	instruction
output	Defines the format of the output document	top-level-element
param	Declares a local or global parameter	top-level-element
preserve-space	Defines the elements for which white space should be preserved	top-level-element
processing-instruction	Writes a processing instruction to the output	instruction
sort	Sorts the output	instruction
strip-space	Defines the elements for which white space should be removed	top-level-element
stylesheet	Defines the root element of a stylesheet	top-level-element
template	Rules to apply when a specified node is matched	top-level-element
text	Writes literal text to the output	instruction

transform	Defines the root element of a stylesheet	top-level-element
value-of	Extracts the value of a selected node	instruction
variable	Declares a local or global variable	top-level-element or instruction
when	Specifies an action for the <choose> element	instruction
with-param	Defines the value of a parameter to be passed into a template	instruction

Exhibit 37: XSLT Functions (from http://www.w3schools.com/xsl/xsl_functions.asp)

Name	**Description**
current()	Returns the current node
document()	Used to access the nodes in an external XML document
element-available()	Tests whether the element specified is supported by the XSLT processor
format-number()	Converts a number into a string
function-available()	Tests whether the element specified is supported by the XSLT processor
generate-id()	Returns a string value that uniquely identifies a specified node
key()	Returns a node-set using the index specified by an <xsl:key> element
system-property	Returns the value of the system properties
unparsed-entity-uri()	Returns the URI of an unparsed entity

Exhibit 38: Inherited XPath Functions (from http://www.w3schools.com/xsl/xsl_functions.asp)

Node Set Functions

Name	Description	Syntax
count()	Returns the number of nodes in a node-set	number=count(node-set)
id()	Selects elements by their unique ID	node-set=id(value)
last()	Returns the position number of the last node in the processed node list	number=last()
local-name()	Returns the local part of a node. A node usually consists of a prefix, a colon, followed by the local name	string=local-name(node)
name()	Returns the name of a node	string=name(node)
namespace-uri()	Returns the namespace URI of a specified node	uri=namespace-uri(node)
position()	Returns the position in the node list of the node that is currently being processed	number=position()

String Functions

Name	Description	Syntax & Example
Concat()	Returns the concatenation of all its arguments	string=concat(val1, val2, ..) **Example:** concat('The',' ','XML') Result: 'The XML'
contains()	Returns true if the second string is contained within the first string, otherwise it returns false	bool=contains(val,substr) **Example**: contains('XML','X') Result: true
normalize-space()	Removes leading and trailing spaces from a string	string=normalize-space(string) **Example**: normalize-space(' The XML ') Result: 'The XML'

starts-with()	Returns true if the first string starts with the second string, otherwise it returns false	bool=starts-with(string,substr) **Example:** starts-with('XML','X') Result: true
string()	Converts the value argument to a string	string(value) **Example:** string(314) Result: '314'
string-length()	Returns the number of characters in a string	number=string-length(string) **Example:** string-length('Beatles') Result: 7
substring()	Returns a part of the string in the string argument	string=substring(string,start,length) **Example:** substring('Beatles',1,4) Result: 'Beat'
substring-after()	Returns the part of the string in the string argument that occurs after the substring in the substr argument	string=substring-after(string,substr) **Example:** substring-after('12/10','/') Result: '10'
substring-before()	Returns the part of the string in the string argument that occurs before the substring in the substr argument	string=substring-before(string,substr) **Example:** substring-before('12/10','/') Result: '12'
translate()	Takes the value argument and replaces all occurrences of string1 with string2 and returns the modified string	string=translate(value,string1,string2) **Example:** translate('12:30',':','!') Result: '12!30'

Number Functions

Name	Description	Syntax & Example
ceiling()	Returns the smallest integer that is not less than the number argument	number=ceiling(number) **Example:** ceiling(3.14) Result: 4
floor()	Returns the largest integer that is not greater than the number argument	number=floor(number) **Example:** floor(3.14) Result: 3
number()	Converts the value argument to a number	number=number(value) **Example:** number('100') Result: 100
round()	Rounds the number argument to the nearest integer	integer=round(number) **Example:** round(3.14) Result: 3
sum()	Returns the total value of a set of numeric values in a node-set	number=sum(nodeset) **Example:** sum(/cd/price)

Boolean Functions

Name	Description	Syntax & Example
boolean()	Converts the value argument to Boolean and returns true or false	bool=boolean(value)
false()	Returns false	false() **Example:** number(false()) Result: 0
lang()	Returns true if the language argument matches the language of the xsl:lang element, otherwise it returns false	bool=lang(language)

not()	Returns true if the condition argument is false, and false if the condition argument is true	bool=not(condition) **Example:** not(false())
true()	Returns true	true() **Example:** number(true()) Result: 1

Exercises

In order to learn more about XSL and stylesheets, exercises are provided.

Answers

In order to learn more about XSL and stylesheets, answers are provided.

Exercises

Stylesheet Chapter => XSL and Style Sheets

Stylesheet Answers => Answers

1. For Chapter 2's exercises, we created an XML document, schema, and stylesheet to define a museum entity. Change that stylesheet to transform that XML document into another XML document, instead of into HTML.
2. In Chapter 4 we created an XML document, schema, and stylesheet to list restaurants, including the most popular, in a city.
    - a. Create a new stylesheet that uses a named template to output the name of the city. Then use two templates with different modes to print out the restaurants, using a larger font for the most popular restaurant.
    - b. Now create a stylesheet that will sort the restaurant names alphabetically, and number them using the `number()` function and a single level of numbering.
    - c. Finally, we will use the `call-template` and `with-param` elements to create a stylesheet that will count the restaurants in a city, and add the prices up, and output it to a text file. This stylesheet will first call a template named "count-restaurants" and will use `with-param` to pass a parameter to count the restaurants. Then it will call a template named "sum-prices" that will use `with-param` to pass the parameter "num" to sum() the upper price for the restaurant.

Stylesheet Chapter => XSL and Style Sheets

Stylesheet Answers => Answers

Answers

Stylesheet Chapter => XSLT and Style Sheets

Stylesheet Exercises => Exercises