

More Advanced XSLT Transformations

TEI@Oxford

July 2011

Feature: number

We can produce numbering based on the position of elements in the text.

- 1 Position within containing element:

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

- 2 Position within whole document:

```
<xsl:template match="p">  
  <xsl:number level="any"/>  
</xsl:template>
```

- 3 Position within an element further up the tree:

```
<xsl:template match="l">  
  <xsl:number level="any" from="lg"/>  
</xsl:template>
```

Understanding automatic numbering

XSL's simple, but powerful, mechanism to number objects, `<xsl:number>`, is **based on their position in the input**. By default this is the *sibling count*. So with this input

```
<list>
  <item>cats</item>
  <item>dogs</item>
  <item>birds</item>
</list>
```

use this XSL

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

... result:

```
1: cats  
2: dogs  
3: birds
```

More complex numbering (1)

There are two common variants of numbering. Firstly, getting the position of the object by its position amongst like-named objects anywhere in the file. So

```
<xsl:template match="figure">  
  <xsl:number level="any"/>  
</xsl:template>
```

produces a number for the `<figure>` element according to its position relative to all other `<figure>` elements.

More complex numbering (2)

Secondly, getting a hierarchical number for an object, when an object is nested deeply. So from

```
<div>
  <div>
    <div>
      <p>Third level division</p>
    </div>
  </div>
  <div>
    <div>
      <div>
        <div/>
        <div>
          <p>Fifth-level division</p>
        </div>
      </div>
    </div>
  </div>
</div>
```

... with this XSL ...

```
<xsl:template match="div">  
  <xsl:number level="multiple"/>  
  <xsl:apply-templates/>  
</xsl:template>
```

... result

```
1
  1.1
    1.1.1
      Third level division
  1.2
    1.2.1
      1.2.1.1
        1.2.1.1.1

        1.2.1.1.2
          Fifth-level division
```


Numbering format

You can also change the format of numbering, using the format attribute of `<xsl:number>`.

1	3
i	iii
I	III
A	C

Example of numbering format

```
<xsl:template match="front/div">
  <h2>
    <xsl:number format="i"/>:
    <xsl:value-of select="head"/>
  </h2>
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="body/div">
  <h2>
    <xsl:number format="1"/>:
    <xsl:value-of select="head"/>
  </h2>
  <xsl:apply-templates/>
</xsl:template>
```

Feature: explicit text nodes

Up until now, we have just put spaces and words into the output, by putting them outside the XML elements. You can make it more explicit by using `<xsl:text>`:

```
<xsl:template match="div">
  <xsl:text> * </xsl:text>
  <xsl:value-of select="head"/>
</xsl:template>
```

rather than

```
<xsl:template match="div">
  * <xsl:value-of select="head"/>
</xsl:template>
```

because the latter also includes the line-ending before the `*`, which can be confusing when you really just want a single space.

Using text

In a previous example, we had

```
<xsl:template match="front/div">
  <h2>
    <xsl:number format="i"/>:
    <xsl:value-of select="head"/>
  </h2>
</xsl:template>
```

But how would we get just a *space* after the number? Space between XML elements is discarded. So we use `<text>`:

```
<xsl:template match="front/div">
  <h2>
    <xsl:number format="i"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="head"/>
  </h2>
</xsl:template>
```

Feature: sort

Let's summarize Punch sections, *sorting* them by heading. We do this by adding `<sort>` elements as children of `<for-each>` or `<apply-templates>`. The `@select` attribute says what to use as sort key.

```
<xsl:template match="body">
  <ul>
    <xsl:for-each select="div">
      <xsl:sort select="head"/>
      <li>
        <xsl:value-of select="head"/>
      </li>
    </xsl:for-each>
  </ul>
</xsl:template>
```

This is a common use-case for `<for-each>`, though we can also use `<sort>` as a child of `<apply-templates>`.

Sorting details

By default, `<xsl:sort>` sorts alphabetically. If you select an element that is numerical, you may get unwanted behaviour.

For example, if the element you are sorting on has values in the range 1 to 100, you will end up with the order 1, 10, 100, 11, 12, ..., 19, 2, 20,

You can tell XSLT that the element has numerical values using a `data-type` attribute:

```
<xsl:template match="listPerson">
  <xsl:apply-templates select="person">
    <xsl:sort select="age" datatype="number" or-
der="descending"/>
  </xsl:apply-templates>
</xsl:template>
```

Feature: @mode on <template> and <apply-templates>

You can process the same elements in different ways using modes:

```
<xsl:template match="/">
  <ul>
    <xsl:apply-templates select="./div" mode="toc"/>
  </ul>
</xsl:template>
<xsl:template match="div" mode="toc">
  <li>
    <xsl:value-of select="head"/>
  </li>
</xsl:template>
<xsl:template match="div">
  <xsl:apply-templates/>
</xsl:template>
```

This is a very useful technique when the same information is processed in different ways in different places.

Feature: variable

Sometimes you want to store some information in a variable for re-use several times. This is another occasion when AVT notation is used:

```
<xsl:template match="entry">
  <xsl:variable name="n">
    <xsl:number/>
  </xsl:variable>
  Entry <xsl:value-of select="$n"/>
  <a name="P{$n}"/>
  <xsl:apply-templates/>
</xsl:template>
```


Feature: output

`<xsl:output>` is a top-level command which determines what the output result is like.

It has a set of attributes:

<code>method</code>	<code>xml html text</code>	type of output
<code>encoding</code>	<i>string</i>	encoding method
<code>omit-xml-declaration</code>	<code>yes no</code>	
<code>doctype-public</code>	<i>string</i>	DTD PUBLIC declaration
<code>doctype-system</code>	<i>string</i>	DTD SYSTEM declaration
<code>indent</code>	<code>yes no</code>	pretty-print result

Lookup tables (1)

When XML is behaving like a database, it make sense to avoid repeating the same phrase. So instead of of saying

```
<condition>good</condition>
```

many times in a document, it is more sensible to say

```
<condition corresp="#c_2"/>
```

and have a lookup table where the conditions are listed:

```
<taxonomy>
  <category xml:id="c_1">
    <desc>excellent</desc>
  </category>
  <category xml:id="c_2">
    <desc>good</desc>
  </category>
  <category xml:id="c_3">
    <desc>reasonable</desc>
  </category>
</taxonomy>
```

Lookup tables (2)

From

```
<condition corresp="#c_2"/>
```

we want to generate the phrase *good condition*. A simple template would be:

```
<xsl:template match="condition">
  <xsl:variable name="this">
    <xsl:value-of select="substring-after(@corresp,'#')"/>
  </xsl:variable>
  <xsl:value-
of select="//taxonomy/category[@xml:id=$this]"/>
  <xsl:text> condition</xsl:text>
</xsl:template>
```

but this is inefficient, as every lookup will result in a quite serious tree traversal. Particularly expensive are XPath queries with noticeable use of `//`, especially at the start of the pattern.

Lookup tables (3)

A better solution is XSLT *keys*, which let the processor maintain the lookup table for us. First we declare a key table, at the top of the stylesheet outside any templates:

```
<xsl:key name="CATS" match="category" use="@xml:id"/>
```

This says that the processor must make an index of the location of all `<category>` elements, based on their `xml:id` attributes. Now we can access the index in a template as follows:

```
<xsl:template match="condition">
  <xsl:value-of
    select="key('CATS',substring-after(@corresp,'#'))"/>
  <xsl:text>condition</xsl:text>
</xsl:template>
```

which is much more efficient, and more readable.

<xsl:key>

Three attributes:

- **name**, the label for the lookup table, in quotes
- **match**, which says which elements to look at (can have wildcards and predicates)
- **use**, which determines the value under which the element will be indexed; this is an XPath expression.

Lookup tables (4)

The XSLT key tables can index *multiple elements* under one key. So

```
<xsl:key name="POINTERS" match="ptr" use="@target"/>
```

will maintain an index of all `<ptr>` elements, with one entry per unique value of the target attribute; when you access it, you will get multiple nodes back.

We could retrieve all the occasions when an `<ptr>` refers to the target 'http://www.bbc.co.uk/' with

```
<xsl:for-each  
  select="key('POINTERS', 'http://www.bbc.co.uk/')"> There  
  is a pointer to the BBC from the text  
<xsl:value-of select="."/>  
</xsl:for-each>
```

Feature: grouping

XSLT 2.0 has a very powerful feature, `<for-each-group>`, which is like `<for-each>`, but allows you to process similar elements together as a group. It is typically used to summarize things:

```
<xsl:template match="body">
  <xsl:for-each-group select="//head" group-
by="parent::div/@type">
  <h1>
    <xsl:value-of select="current-grouping-key()"/>
  </h1>
  <ul>
    <xsl:for-each select="current-group()">
      <li>
        <xsl:value-of select="."/>
      </li>
    </xsl:for-each>
  </ul>
</xsl:for-each-group>
</xsl:template>
```