Extracted from:

XSLT Jumpstarter

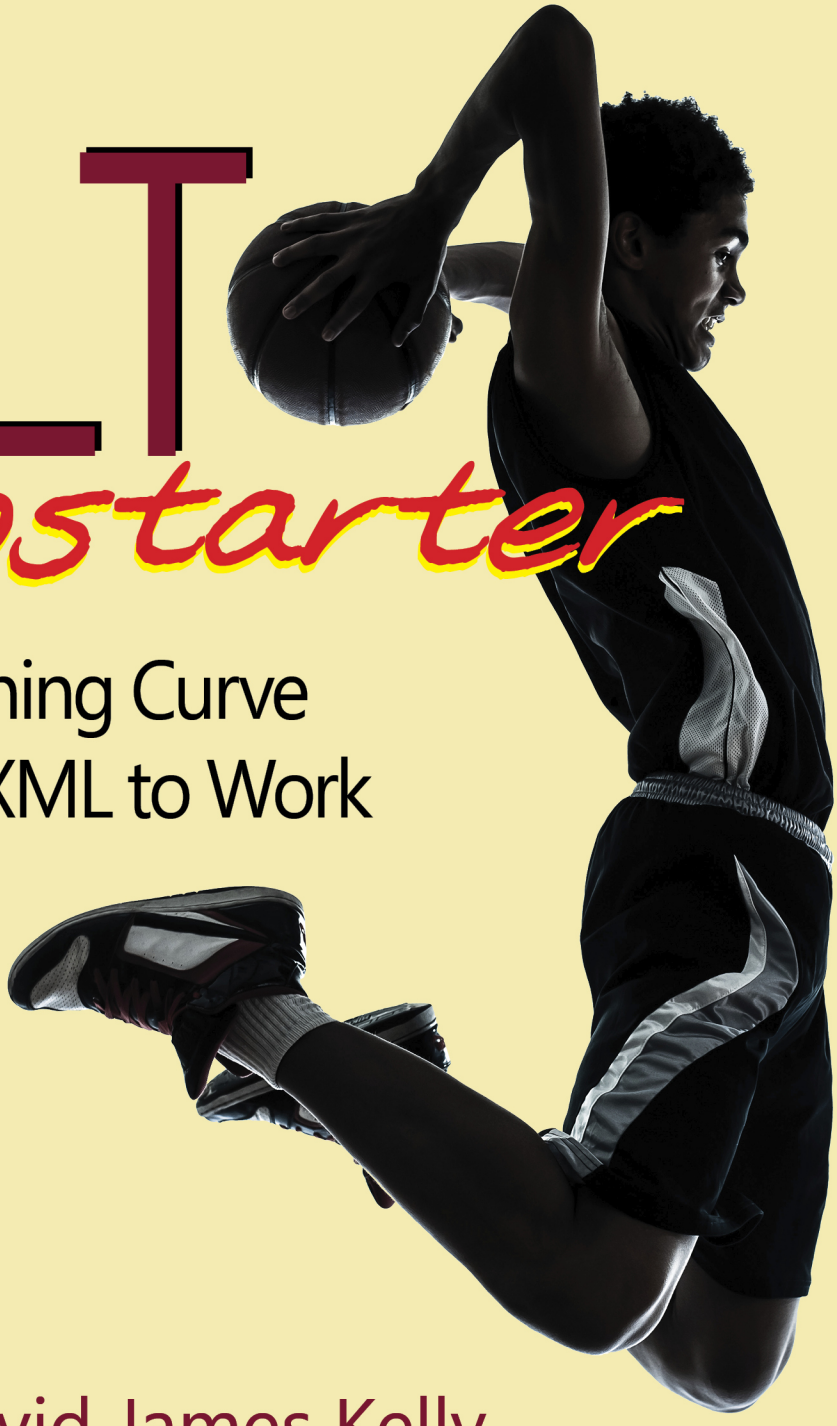
Level the Learning Curve and Put Your XML to Work

Peloria Press

Raleigh, North Carolina

XSLT Jumpstarter

Level the Learning Curve and Put Your XML to Work



David James Kelly

Foreword by Dave Thomas

XSLT Processing: Under the Hood

One way to think about XSLT processing is to think of the XML as a stream of coins or tokens, all sizes and shapes, and the XSLT templates as a series of slots of corresponding sizes and shapes.

Figure 9, *An XML structure matched by XSLT templates*, on page 4 shows the XML as a group of connected objects, essentially a set of connected tokens. The different parts of the XML appear in this diagram with different shapes. At the same time, the XSLT side of the figure contains slots of corresponding shapes that allow the different parts of the XML to "fall through."

A couple of things you'll note immediately:

- The stylesheet doesn't have templates for all of the XML elements.
- The templates in the stylesheet aren't structured the way the XML is structured. In fact, they aren't explicitly connected at all. We'll see the advantage of this arrangement in a moment.

To understand how our example worked, let's see what happens when we start "dumping" the XML into the stylesheet. For this analogy to work, you might think of the XML as being turned upside down, with gravity pulling the `<customer>` tag in first, then everything else following in *document order*. The term document order describes the order the XML is handled by the XSLT processor, and it's worth spending a little time to understand what it means.

The XSLT processor typically parses the XML document into a document tree. The document tree for our sample document would look like the diagram we saw back in Figure 5, *An XML document tree*, on page ?.

In this example, the document tree on the right is formed out of the XML document on the left. The vertical lines define the scope of the parent nodes, and the horizontal lines point to specific children of a parent node. By parent and child, we mean simply that a parent is a tag that contains another tag, and a child is a tag that is contained by a parent tag.

The example shows that attributes are not *child nodes*. (Check out the `id="id_1234567"` attribute for the `<customer>` tag.) They belong to elements, but they aren't children of elements. It sounds like an odd distinction to make, but we'll see in Chapter 7, *XPath: The Sibling Language*, on page ? that it has an effect when we attempt to process the children and attributes of an element.

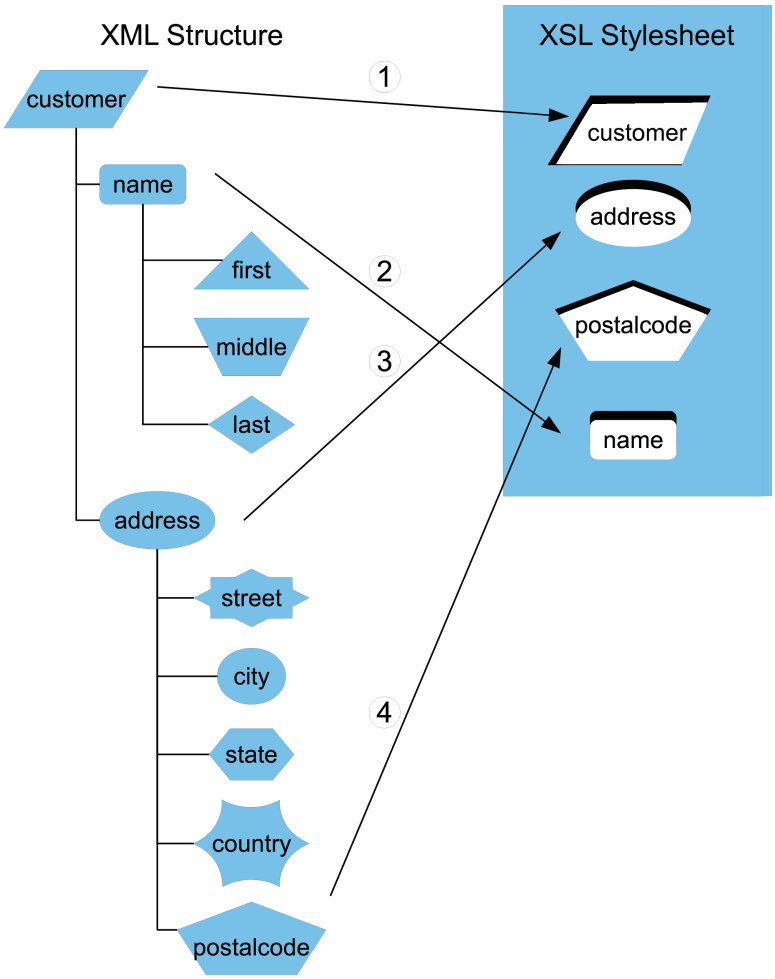


Figure 9—An XML structure matched by XSLT templates

Joe asks:

What's the difference between an element and a node?

We mention elements, attributes, and text quite a bit in this book. In the XML world, all of these things and others are referred to as *nodes*. A node is just a general term for a variety of XML object types.

When we use the word "node" by itself, we can be talking about any or all of these things (including XML comments and XML processing instructions). However, when

we want to be more specific, we'll say "element node" or "text node," or just element and text.

We'll need to keep the differences between *node* and *element* in mind especially once we see the XPath wildcards for each of them. But let's not get ahead of ourselves.

Another point the example shows is that instances of text are XML nodes themselves. Text nodes are children of their containing elements, and their containing elements are their parents. The main difference between an element and a text node is that text can't have any children. Text always occurs at the end-point of the branches of the document tree—which is to say, text is always a leaf of the document tree.

Finally, notice at the top of the document tree on the right that the first node in the document is the root. (How many other trees have their roots on top?) The root node is present in every XML document, and it contains everything else, including the root element (in this case, <customer>).

The nodes are processed in the order they are presented in the tree part of the diagram, working from top to bottom. In document order, the order of processing is from the top branch down and to the right as far as possible, then back up to the next-topmost, unprocessed element in the tree, and so forth. Referring to the right side of Figure 5, *An XML document tree*, on page ?, the nodes are processed from the top line to the bottom line, one line at a time, in the order shown.

If the document were processed in hierarchical order, the root element in the document (<customer>) would be processed first, followed by the two tags at the next level in the document (<name> and <address>), followed by the third-level tags, and so forth. But that's not how it works.

It seems too sensible to be true, but every now and then there is a reassuring sense of order in XSLT.

Now let's walk through the process in detail, with our XML being dumped into the stylesheet in document order:

1. The document root and <customer> tag come first, and immediately we find a template that matches both in the stylesheet. In the XSLT, this template looks like this:

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

Great! With this match, we have changed the context for execution to the <customer> tag. The rest of the XML "falls through" into the matching template and gets processed according to the instructions inside the template. In this case, the instructions are <xsl:apply-templates>, so the XSLT processor looks for templates to match whatever comes next in the XML document.

2. Next, the processor comes to the <name> tag and finds a template that matches it:

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

Also fine—we didn't want any of the name in our report, so this template is empty. We don't do any more processing, and the <first>, <middle>, and <last> tags are forever after ignored.

3. The XML keeps streaming along, and next up is the <address> tag. Yep, we find a template that fits, so we do whatever the template says:

```
<xsl:template match="address">
  <xsl:apply-templates select="postalcode"/>
</xsl:template>
```

The template for <address> has an instruction to apply templates, but it includes a qualifier, the select="postalcode". This instruction tells the XSLT processor to go off looking for templates again, but this time, it will only look for a template that matches <postalcode>.

Since we're only interested in the postal code, we use the select= attribute to narrow the scope of what will be processed inside the <address> tag. If we left off the select="postalcode" attribute, the XSLT processor would apply templates for all of the XML that it finds inside the <address> tag.

You can try this yourself. Going back to the stylesheet in code on page ?, delete (or comment out with the <– –> notation) the template that matches on <address>, then process the XML file again. (If your browser is still up from the first time you ran it, and the result is still visible, you can just refresh the browser.) Do you see what happens? The text goes from showing just the postal code to showing the text of the entire address.

(Be sure to undelete or uncomment the template for some of the following discussions.)

4. We sent the processor off to match against the <postalcode> tag, and now we've found it. Our stylesheet does only one thing: return the value of the <postalcode> tag. The <xsl:apply-templates> in the template for <postalcode>

returns the text for us—we'll get to how that works in a short while. In the meantime: mission accomplished. We have the value of the postal code from the file.

That's the general outline of how XSLT processing works: match some XML with a template, do some work inside the template, then get instructions inside the template for what to do with the next bit of XML. It sounds simple enough, but it can do a lot of complex work with this processing pattern.

The trick here is that we don't necessarily need to know how many of a given tag there are in the XML, or in what order they occur. Because of the <xsl:apply-templates> tag, and because the templates aren't explicitly connected to each other, the XSLT processor just takes things in the order they come, and does its best to find templates to match. If it finds a match, it follows the instructions then goes to the next piece of XML; if it doesn't, the processing stops following the branch at that point in the structure of the XML, then goes on to whatever branch in the XML comes next. It's this flexibility to follow XML wherever it goes that gives XSLT a big part of its power.