


<oo> → <dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com) 

Last modified: 2018-03-16T22:34:07+0000

XSLT identity transformation

The XSLT *identity transformation* is used to transform an XML document to itself, that is, to generate XML output that is identical to the XML input. By itself this is not very useful, since there are more computationally efficient ways to produce an identical copy of a document. Where it pays off, though, is if you want to make an *almost* identical copy, except that you want to introduce a small but systematic change or two. You can do this by using the identity template to transform everything in the document except the parts that you want to modify. The result is that you produce a copy of the document that is identical to the original except that it includes your modifications.

For example, you might have a document filled with sonnets with a structure like:

```
<sonnet number="I">
  <line>From fairest creatures we desire increase,</line>
  <line>That thereby beauty's rose might never die,</line>
  <line>But as the riper should by time decease,</line>
  <line>His tender heir might bear his memory:</line>
  <line>But thou contracted to thine own bright eyes,</line>
  <line>Feed'st thy light's flame with self-substantial fuel,</line>
  <line>Making a famine where abundance lies,</line>
  <line>Thy self thy foe, to thy sweet self too cruel:</line>
  <line>Thou that art now the world's fresh ornament,</line>
  <line>And only herald to the gaudy spring,</line>
  <line>Within thine own bud buriest thy content,</line>
  <line>And tender churl mak'st waste in niggarding:</line>
  <line>Pity the world, or else this glutton be,</line>
  <line>To eat the world's due, by the grave and thee.</line>
</sonnet>
```

that you'd like to convert to:

```
<sonnet>
  <number>I</number>
  <line>From fairest creatures we desire increase,</line>
  <line>That thereby beauty's rose might never die,</line>
  <line>But as the riper should by time decease,</line>
  <line>His tender heir might bear his memory:</line>
  <line>But thou contracted to thine own bright eyes,</line>
  <line>Feed'st thy light's flame with self-substantial fuel,</line>
  <line>Making a famine where abundance lies,</line>
  <line>Thy self thy foe, to thy sweet self too cruel:</line>
  <line>Thou that art now the world's fresh ornament,</line>
  <line>And only herald to the gaudy spring,</line>
  <line>Within thine own bud buriest thy content,</line>
  <line>And tender churl mak'st waste in niggarding:</line>
  <line>Pity the world, or else this glutton be,</line>
```

```
<line>To eat the world's due, by the grave and thee.</line>
</sonnet>
```

That is, your input has a **@number** attribute that you'd like to replace with a **<number>** child of the **<sonnet>** element, but you'd like to keep the wrapper **<sonnet>** element and the internal **<line>** elements. To make that change you can start with an identity transformation, which applies templates to every node in the document and tells it to reproduce itself unchanged in the output, except that you also write a template that handles the numbering specially.

The identity template looks like:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

The preceding is an entire XSLT stylesheet that transforms a document to itself unchanged. The way it works is that it matches all attributes (**@***) and all other nodes (**node()**; this includes the document node and all elements, **text()** nodes, comments, etc.), makes a copy of those, and then applies templates to any attributes or other nodes associated with whatever it's processing at the moment. Since an XSLT transformation starts automatically at the document node (and because the document node is a node, this template will match it), you wind up processing everything in the document, moving down level by level through the hierarchy. Because **text()** nodes are nodes, too, this also winds up copying all of the text.

There are two subtle details that enable such a simple template to do so much work:

1. The **<xsl:copy>** element makes a *shallow copy*. This means that it creates a copy of the node that it is processing, but it doesn't do anything with any attributes or child elements or textual content. For example, if you apply **<xsl:copy>** to a **<sonnet>** element in the example above, it will create a **<sonnet>** element in the output, but it won't copy the **@number** attribute or the **<line>** child elements. This is why we need to apply templates explicitly inside the **<xsl:copy>** tags.
2. We have to apply templates to both nodes and attributes. We don't usually think of it this way, but because XPath looks by default at the child axis, when we apply templates to **node()**, we're applying templates to all nodes *on the child axis* of the current context (that is, of the element we matched in the template that is doing the work at the moment). In other words, **<xsl:apply-templates select="node()" />** is synonymous with **<xsl:apply-templates select="child::node()" />**. Since *attributes are on the attribute axis, and not on the child axis*, this would not apply templates to attributes, which means that attributes on elements in the input document would be lost during the transformation. To avoid losing them, we have to apply templates to the union (using the union operator **|**) of anything on the attribute axis (**@***) and any node on the child axis (**node()**).

One more bit of magic is that XSLT templates have *precedence* rules, which specifies what happens when more than one template matches a node that is being processed, and we exploit those rules to override the identity template when we want to change something during the transformation. The most important precedence rule is that *the more specific @match value wins*. Since the **@match** value of the identity template, above, is very general (it matches any attribute and any other node), just about any other **@match** value would be more specific. What we'll do for our present task, then, is let the identity template take care of everything in our document except the bits that we want to change. The identity template will match

absolutely everything in the input document, but our more specific template for what we want to change will override it where we need it to.

This simplified example contains just two element types: `<sonnet>` and `<line>`, but in Real Life you would have other elements: a root element (perhaps `<sonnets>`), some metadata, headers or titles, and perhaps more. We want to leave all of those other element types unchanged, and we also want to leave `<line>` unchanged, but we want to make two changes to `<sonnet>`:

1. We want to remove the `@number` attribute, and
2. We want to add a `<number>` child element.

A full XSLT stylesheet to do that would be:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="@* | node()">
    <xsl:copy>
      <xsl:apply-templates select="@* | node()" />
    </xsl:copy>
  </xsl:template>
  <xsl:template match="sonnet">
    <xsl:copy>
      <number>
        <xsl:value-of select="@number" />
      </number>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

The identity template matches everything in the input document: the document node, all elements, all attributes, and all `text()` nodes. Both templates match `<sonnet>` elements; the second one matches them specifically and the identity template matches them because it matches all nodes. This means that the identity template will be used to make no changes in anything else, but the second template, which has the more specific `@match` attribute value and therefore outranks the identity template in case of a tie, will be the one that gets to handle the `<sonnet>` element. What the sonnet-specific template does is copy the element it just matched (that is, create a shallow copy of the `<sonnet>` element in the output) and then, inside the element it has just created in the output, create a new `<number>` child element, the content of which is the value of the `@number` attribute that was on the original `<sonnet>` we're processing. Below that new `<number>` child element of the `<sonnet>` we apply templates without a `@select` attribute, which means that we apply templates to all of the child nodes of the `<sonnet>` we're processing at the moment. Those child nodes are the `line` elements of the sonnet, and when we apply templates to them, the identity template does the processing and just copies them unchanged to the output.

So what happened to the original `@number` attribute? Attributes are not children because they aren't on the child axis; they're on the attribute axis. This means that in our template that matches `<sonnet>` elements, our `<xsl:apply-templates />` without a `@select` attribute applies templates to all of the *children* of the current context (in this case, the `<line>` elements), but not to its attributes. This means that the original `@number` attribute disappears because we simply don't apply templates to it.

The XSLT 3.0 way

The identity template described above does the job well, and you'll encounter it widely, including in tutorials and homework answers in this course. But since the advent of XSLT 3.0, there is an alternative way of

describing a default identity operation in a single line:

```
<xsl:mode on-no-match="shallow-copy"/>
```

The new XSLT 3.0 **<xsl:mode>** element is a *top-level* element, which means that it's a child of the root **<xsl:stylesheet>** element and a sibling of **<xsl:output>** and **<xsl:template>** elements. What this statement does is overwrite the built-in rules, which would otherwise be that 1) if there's no template for an element, throw away the tags and process its children, and 2) if there's no template that matches **text()** nodes, output the text. This new rule says that if there's no template for any node (element, **text()**, anything else), make a shallow copy of it. It will apply first to the document node, because that's where XSLT starts its work, and it will then work its way down the tree. As with the older strategy described above, you can let this XSLT 3.0 identity rule take care of most of your document, and write explicit rules only for the stuff you want to change. Our solution above could thus be rewritten as:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:mode on-no-match="shallow-copy"/>
  <xsl:template match="sonnet">
    <xsl:copy>
      <number>
        <xsl:value-of select="@number"/>
      </number>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

It's better to use this new XSLT 3.0 version in new stylesheets, but because XSLT 3.0 is fairly young, you'll still run into lots of examples of the older strategy. Don't forget, though, that if you use the XSLT 3.0 method, you should set the value of the **@version** attribute on the **<xsl:stylesheet>** root element to **3.0** and you should select Saxon-PE or Saxon-EE as your transformation engine. If you don't do that, you won't raise an error in <oXygen/>, but it's still a mistake; if you're using XSLT 3.0 features, set the value of the **@version** attribute accordingly and use one of the XSLT 3.0 transformation engines.