

DITA 1.3 Feature Article
A Brief Introduction to XSL for Processing DITA
Content
An OASIS DITA Adoption Technical Committee Publication

Contents

Chunk1739435240

Part

I

What is XSL?

If you have ever transformed DITA content into HTML, Web help, a PDF or anything else, you have used XSL. If everything went well, you probably didn't think too much about how the transformation happened. If things did not go well, and the output was not what you expected, or was not there at all, you probably gave the transformation a little more (unkind) thought and either passed the problem off to someone else on your team or rolled up your sleeves to have a look under the hood to examine the workings of the DITA Open Toolkit (DITA OT). The DITA OT is a collection of Java files, scripts, build files, and some other stuff, but its heart is the XSL code used to transform your DITA content. This article first focuses on the basics of understanding XSL and then turns to a couple of examples of XSL within the DITA OT, specifically for transforming DITA content.

~~The simplest way to describe XSL is that it transforms DITA, or any other kind of XML, into something else like HTML, Web help or a PDF.~~

XSL is a general term that is the base for a set of XML transforms used for different purposes. The acronym stands for eXtensible Stylesheet Language. XSL is the basis for the following:

- XSLT, or **XSL Transformations**. XSLT is a language for transforming XML documents.
- XPath, or the **XML Path Language**. XPath is an expression language used by XSLT to access or refer to parts of an XML document.
- XSL-FO, or **XSL Formatting Objects**. XSL-FO is an XML vocabulary for specifying formatting semantics.

Informally, “XSL” often refers to XSLT only. In turn, “XSLT” often refers informally to both XSLT itself and XPath, since most XSLT documents use XPath expressions. XSLT is the focus of this article.

These definitions are from the World Wide Web Consortium (W3C) (<https://www.w3.org>), which is an international community that develops open standards to ensure the long-term growth of the Web. Much like OASIS (Organization for the Advancement of Structured Information Standards, <https://www.oasis-open.org/>) develops and maintains the specifications for DITA, the W3C develops and maintains the specifications for HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), XSLT, and other Web-related standards. Like DITA, the XSL technologies are open standards, and so their workings are transparent, platform independent, and have a robust architecture.

How does XSLT turn your XML content into output?

Before talking specifically about the stylesheets in the DITA OT, which are a bit of a special case, it is important to have a general understanding of how XSLT works. The first thing to understand is that XSLT doesn't turn XML into something else by itself. A stylesheet is just a static file. However, the stylesheet gives instructions to an XSLT processor such as Xalan (<https://xalan.apache.org/>) or Saxon (<http://saxon.sourceforge.net/>) and the processor uses those instructions to turn XML into HTML or PDF or whatever you choose.

If you have ever used a desktop publishing application, you are familiar with templates. In the desktop publishing world, a template usually includes page definitions, paragraph and character formats, table formats, and so forth. As you edit your document, you apply styles based on your knowledge of how the text should look.

When you are transforming XML, you do not have the opportunity to apply specific styles to the text manually; you are depending on an automatic process to do that. You have to supply the process with the correct instructions, such as

"When you encounter a title element, check to see if it is part of a topic that is nested in another topic. If so, indent it and make it bold and 14 points. If not, don't indent it and make it bold and 18 points."

~~These kinds of instructions are found in XSL stylesheets. Each stylesheet contains a number of templates, most of which are designed to process a specific element, as we are going to see shortly. Like those templates, the stylesheet templates test an element for its context (because some elements should be processed differently depending on where they occur) and apply formatting to the element.~~

The XSLT processors mentioned above iterate through the input XML and as they encounter a specific element, they find the matching template and follow the instructions in that template to create a corresponding HTML element with appropriate formatting attributes (or better, formatted by an accompanying CSS) or an XSL-FO element with appropriate formatting attributes.

This is a very general explanation. If you're curious to know more, the DITA OT documentation fully explains the whole process. This article is more concerned with explaining how XSLT itself transforms XML content to an output format, so let's move on to that.

Anatomy of an XSLT File

Here is very simple, non-DITA, XML file—a contact list (**contactlist.xml**):

```
<addressbook>
  <entry>
    <fname>Llewellyn</fname>
    <lname>Jones</lname>
    <phone type="home">+1 203-555-1212</phone>
  </entry>
  <entry>
    <fname>Fiona</fname>
    <lname>Murphy</lname>
    <phone type="work">+1 414-555-9876</phone>
    <phone type="mobile">+1 414-555-4321</phone>
  </entry>
  <entry>
    <fname>Angus</fname>
    <lname>MacPherson</lname>
    <phone type="mobile">+1 850-555-4783</phone>
  </entry>
</addressbook>
```

Figure 1: addressbook.xml

The goal is to transform the contact list into a HTML file that looks like this:

<p>Jones, Llewellyn</p> <ul style="list-style-type: none">• +1 203-555-1212 (home) <p>Murphy, Fiona</p> <ul style="list-style-type: none">• +1 414-555-9876 (work)• +1 414-555-4321 (mobile) <p>MacPherson, Angus</p> <ul style="list-style-type: none">• +1 850-555-4783 (mobile)
--

Figure 2: addressbook.html

Here's an XSLT stylesheet that will give you those results:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
3
4   <xsl:template match="addressbook">
5     <html xmlns="http://www.w3.org/1999/xhtml">
6       <head>
7         <title>My Contact List</title>
8       </head>
9       <body>
10        <xsl:apply-templates select="entry"/>
11      </body>
12    </html>
13  </xsl:template>
14
15  <xsl:template match="entry">
16    <p>
17      <b>
18        <xsl:apply-templates select="lname"/>
19        <xsl:text>, </xsl:text>
20        <xsl:apply-templates select="fname"/>
21      </b>
22    </p>
23    <ul>
24      <xsl:apply-templates select="phone"/>
25    </ul>
26  </xsl:template>
27
28  <xsl:template match="fname">
29    <xsl:value-of select="."/>
30  </xsl:template>
31
32  <xsl:template match="lname">
33    <xsl:value-of select="."/>
34  </xsl:template>
35
36  <xsl:template match="phone">
37    <li>
38      <xsl:for-each select=".">
39        <xsl:value-of select="."/>
40        <xsl:text> <xsl:text>
41        <xsl:value-of select="@type"/>
42        <xsl:text></xsl:text>
43      </xsl:for-each>
44    </li>
45  </xsl:template>
46
47 </xsl:stylesheet>

```

Figure 3: addressbook.xsl

It might look complicated, but don't panic! We will walk through each line of this stylesheet.

Line 1 is the standard XML declaration found at the beginning of every XML or XSLT file. Line 2 defines the root element of the stylesheet. XSLT conforms to the same syntax rules as XML—it is itself XML after all—so it must have a single root element. Notice this root element matches the root element of the XML file we're processing.

Line 4 begins the first template. Every stylesheet template begins with `<xsl:template>`. Stylesheets usually include multiple templates. Each template usually processes one specific XML element, although some templates do other things, like create variables for use elsewhere. How does each template know which element it's supposed to process? The `@match` attribute. The first template is set to match on the `<addressbook>` element, which is the root element of `contactlist.xml`. When you're processing a single XML file, this is generally where you start...at the root. And from there, you move along the axes of your XML structure. More on that in a minute. For right now, let's concentrate on where we are...at the root, or `<addressbook>` element.

To transform this XML content to HTML it needs to match the expected structure for the HTML file. Think of it as creating “buckets” to “pour” the appropriate XML content into. For this simple contact list example, the basic HTML structure will be:

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <p><b></b></p>
    <ul>
      <li></li>
    </ul>
  </body>
</html>
```

Each person’s name will be wrapped in a `<p>` (paragraph) element and made bold by the `` element, with each phone number in a `` (list item), all gathered together in a `` (unordered list). Ultimately, the `<p>`, ``, ``, and `` elements are going to be created by other templates specific to the `<fname>`, `<lname>`, and `<phone>` elements in the original XML file. But the overall structure needs to be created just once, at the very start of processing. It makes sense to create that structure in the very first template, or the one that processes the root element `<addressbook>`. That’s what lines 5-9 and 11-12 do.

All the content of the XML file—everything in the `<fname>`, `<lname>`, and `<phone>` elements—is going to go inside the body of the resulting HTML file. We just said that those other elements are going to be processed by other templates. You can’t create a template inside another template, but you can *call* a template (created somewhere else) from within another template. That’s what line 10 does. `<xsl:apply-templates>` says: “Put the result of some other template’s processing right here.” more specifically, line 10 says “Put whatever the result of the entry template is here, inside body.” The `@select` attribute on `<xsl:apply-templates>` specifies which template to use. The `@select` attribute is found on many XSLT elements and it selects a specific template or value to use at that moment.

Line 13 closes the first template. Line 15 starts a new template and this one processes the `<entry>` element in the `contactlist.xml` file. In this template, we create the `<p>` and `` “buckets” into which we’re going to pour the contents of the `<fname>` and `<lname>` elements. We also create the `` element that will contain the `` elements that in turn hold the contents of the `<phone>` elements.

```
<p><b>[last name], [first name]</b></p>
```

First we want to pour the contents of the entry’s `<lname>` element, then we want to pour the contents of the entry’s `<fname>` element. (Don’t worry about the comma for now.) Line 16 creates the `p` element and line 17 creates the `b` element that contain the first and last name.

Just like we did earlier when we needed the results of one template inside of another template, we want to use an `<xsl:apply-templates>` to call the template that processes `<lname>` and then another `<xsl:apply:templates>` to call the template that processes `<fname>`. Lines 18 and 20 do that.

Coming back to the comma between the last and first names, any time you need to dynamically insert text that isn’t part of the actual XML content, you can use `<xsl:text>` to do that. It’s very handy. That’s line 19. Lines 21 and 22 close the `` and `<p>` elements.

Following the paragraph with the first and last name, we want to create an unordered (bulleted) list of the contact’s phone numbers. Line 23 opens the `ul` and line 25 closes it. Each phone number is going to end up being an `` inside of the ``. However, just like with `<fname>` and `<lname>`, we’re going to have a separate template to process the `<phone>` elements, and it’s in that template that we will create the line items. Line 24 calls the “phone” template that we’re going to create. Line 26 closes the “entry” template.

Now that we’ve applied a template that processes `<lname>`, another that processes `<fname>`, and a third that processes `<phone>`, we need to create those templates. Lines 28-30 form the “fname” template, which is a very simple template. Remember that this template is already being called inside a `` element so no further markup is

necessary—we just want the text content of the `<fname>` element inside of the `` element. The “fname” template specifies that it’s looking at an `<fname>` element so need to say, essentially, “select the content of this element itself.” Line 29, `<xsl:value-of select="."/>`, does that. The “.” is an XPath expression meaning “self.” The “lname” template (lines 32-34) works exactly the same way.

Let’s stop here briefly and talk about XPath. What is it and what is it used for? In short, it’s how we navigate through an XML tree and select the element or node we want. We’re familiar with XML parent and child elements, sibling elements, and so forth. If we represent the contact list XML as a tree, we get:

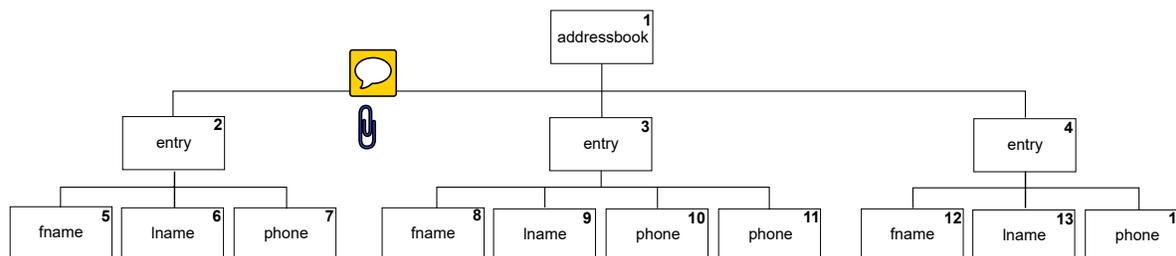


Figure 4: addressbook.xml tree

Each of the boxes is an XML element, of course, but also an XPath node. In our simple XSLT we started at the top of the tree, `<addressbook>`, by beginning with `<xsl:template match="addressbook">` (line 4). From there, we navigated straight “down” into the XML tree to the `<entry>` node by using `<xsl:template match="entry">` (line 15). Once inside entry, we then navigated “down” to the `<fname>` and `<lname>` nodes with the additional `<xsl:template match..>` statements (lines 28 and 32).

Note: Understand that “down” and “up” are just convenient ways of describing movement though the XML structure as represented in the previous diagram. The actual XML structure is abstract and doesn’t have any literal directionality.

Think of this as moving through a family tree. All the relationships depend on the point of view of the element or node you’re working with *at the moment*. If we’re working with addressbook (that is, we’re in a template that processes addressbook), then from addressbook’s point of view, it has three children, each named entry. If we’re working with addressbook but we need a piece of information from the first phone child of the second entry node, we need to navigate from addressbook to the second entry node and from there to its second phone node: path 1-3-10.

Fortunately, XPath exists exactly for navigating paths between the current node and its parents, siblings, children, and so forth. XSLT uses XPath extensively. Our simple contact list XSLT includes only one XPath expression, the “.” expression mentioned earlier, but there are a lot more.

Returning to our 1-3-10 example, we would use XPath like this for that navigation:

```
<xsl:value-of select="entry[2]/phone[2]">
```

We don’t have to include addressbook in the path because we’re already there, remember—we’re in a template that processes addressbook. We want to navigate to its entry children and specifically to the second one, which is what the `[2]` predicate indicates. From that second entry node, we want to navigate down into its children. The `/` (slash) after entry indicates a child of entry. The phone indicates that we’re looking specifically at phone child nodes of entry and more specifically, at the second one, also indicated by the `[2]` predicate.

That’s a lot to take in, but it’s not important to fully understand it right now. Just understand that XPath allows you to move from parent to child, from child to parent, from sibling to sibling, to test from a specific node all the way “up” the tree for a specific ancestor or all the way “down” the tree for a specific descendant, and much more. Let’s leave the subject of XPath for now. We’ll talk more about them shortly, when we look at templates from the DITA OT. For now, just understand what they are.

Let’s finish dissecting our stylesheet. We still need a template to process `<phone>` elements. Lines 36-45 create that template. It’s a little more complex than the “fname” and “lname” templates. Here’s how it works.

Lines 36 and 45 start and end the template, of course. We know that the contents of each `<phone>` element go in an `` element. The `` that contains all of the `` elements has already been created by the “entry” template (lines 23-25). Lines 37 and 44 begin and end the ``.

Line 38 contains something we haven’t seen before: `<xsl:for-each select=".">`. We used `<xsl:value-of select="...">` previously to get the contents of `<fname>` and `<lname>`. However, notice that there can be more than one `<phone>` element for an entry and we need to loop through and get them all. That’s what `<xsl:for-each...>` does. Line 43 ends the for-each loop.

Within the for-each loop, we get back on familiar ground with line 39, `<xsl:value-of select="."/>`, which selects the contents of the `<phone>` element currently being processed by the loop. Following the phone number, we know that we want the value of the `@type` attribute enclosed in parentheses. The parentheses aren’t part of the XML; we’re adding them on the spot, so we can use `<xsl:text>` for the opening and closing parentheses (lines 40 and 42).

Line 41 selects the value of the `@type` attribute on the `<phone>` element currently being processed. Finally, line 47 ends the stylesheet.

If your head is spinning a little, don’t worry. Here are your most important takeaways:

- Stylesheets follow the same syntax rules as XML because they *are* XML.
- Stylesheets simultaneously create a structure (HTML or XSL-FO), traverse an XML tree, and place the contents of elements and their attributes into that structure.
- Stylesheets usually contain multiple templates.
- Each template ideally processes one element.
- Some templates produce output and some simply process an element for use elsewhere.
- Stylesheets use XPath to traverse an XML tree, using many different operators to select children, parents, siblings, self, etc.

The DITA OT stylesheets

Now that you’ve seen a basic XSL stylesheet, let’s look specifically at stylesheets in the DITA OT.

When you process, or build output for a ditamap, you send it to the DITA OT. The DITA OT consists mainly of a bunch of build files and stylesheets. (It contains other important files as well, but they don’t matter for this article.) These build files and stylesheets are grouped together into plugins. Generally, each plugin corresponds to a particular output type. When you specify that you want XHTML/HTML output, or Webhelp, or a PDF, or any other kind of output, you (or the tool you are using) sends an instruction to the DITA OT telling it which plugin to use in order to process the content you are sending to it. The specified plugin could be one of the default plugins that come with the DITA OT, or it could be a custom plugin that you or someone else developed specifically for your outputs.

Before we go into a discussion of the stylesheets, you should understand that there are a lot of things that happen when you first start a build. Your content is validated, your map is parsed (and filtered, if applicable), keys and conrefs are resolved, and so forth. All of that is beyond the scope of this article. Here, we’re going to focus just on how the stylesheets in the applicable plugin turn content from DITA XML to XHTML or to a PDF.

To keep things simple for this article, we’ll look at only two of the DITA OT plugins: the one that generates XHTML (**org.dita.xhtml**) and the one that generates PDF (**org.dita.pdf2**).

If you’re creating XHTML-based output, the DITA OT outputs one XHTML file per DITA topic (unless you specify otherwise). It’s sufficient to transform the DITA XML directly to XHTML and that’s mostly what the templates in the **org.dita.xhtml** plugin do.

If you’re creating PDF output, the process is a little more complex. You don’t (usually) want a separate PDF file for each topic; you want one PDF for the entire map. The first step, then, is to merge the entire map into one big DITA XML file. That merged file is then sent forward for processing. To create a PDF from DITA XML, you need a separate application called a PDF renderer. PDF renderers do not understand DITA XML, so the next step is to transform the DITA XML into another form of XML called XSL-FO. This is what most of the templates in the **org.dita.pdf2** plugin do. The resulting XSL-FO file is then sent to the PDF renderer and it creates a PDF.

You might guess at this point that the templates in **org.dita.xhtml** and **org.dita.pdf2** are going to be rather different because they are doing different kinds of transforms—DITA to XHTML and DITA to XSL-FO. You'd be correct, although they all have one important thing in common: they all use XSLT and XPath.

The simple addressbook example we looked at consisted of a single stylesheet. You'll quickly notice that each of the plugins in the DITA OT consists of multiple stylesheets because it would be far too impractical to include hundreds of templates in a single stylesheet. A stylesheet can import or include other stylesheets so that they all end up functioning as if they were a single stylesheet.

An example of a DITA to XHTML template

Let's start by looking at the "codeblock" template from the XHTML plugin. This template is found in the file **pr-d.xsl**, which is found in the `xsl\xslhtml` folder of the **org.dita.xhtml** plugin.

If you look in the folder, you'll see that there are several stylesheets there. There are separate stylesheets for the domains (**hi-d.xsl**, **pr-d.xsl**, **ui-d.xsl**, etc.) as well as stylesheets specific to each topic type (**conceptdisplay.xml**, **refdisplay.xml**, **taskdisplay.xml**). There are some other miscellaneous specific stylesheets (**tables.xml**, **glossdisplay.xml**, **rel-links.xml**, **syntax-braces.xml**, etc.) and some map-specific stylesheets (**map2TOC.xml**, **mapwalker.xml**). Finally, there is the "grab-bag" stylesheet **dita2htmlImpl.xml**, which processes most of the other elements not processed by the more specific stylesheets.

All the DITA OT plugins contain multiple stylesheets, (organized differently and with different names, of course). As you work with the various plugins, you'll become accustomed to where to find the templates you need.

The "codeblock" template in the XHTML plugin looks like this:

```

14 <xsl:template match="*[contains(@class, ' pr-d/codeblock ')]" name="topic.pr-d.codeblock">
15   <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-startprop ')]"
      mode="out-of-line"/>
16   <xsl:call-template name="spec-title-nospace"/>
17
18   <pre>
19     <xsl:call-template name="commonattributes"/>
20     <xsl:call-template name="setscale"/>
21     <xsl:call-template name="setidaname"/>
22
23     <code>
24       <xsl:apply-templates/>
25     </code>
26   </pre>
27
28   <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-endprop ')]"
      mode="out-of-line"/>
29 </xsl:template>

```

Figure 5: Codeblock template (org.dita.xhtml plugin)

Line 14 begins the template and line 29 ends it. But line 14 is very different from what we saw in the addressbook templates. There, we selected the element by its name (using the `@match` attribute). Here, the `@match` attribute is much more complex: `"*[contains(@class, ' pr-d/codeblock ')]"`. This is the standard way of selecting elements throughout the DITA OT, in all plugins. Every DITA element has a unique `@class` attribute value. To select an element, you match on its class. The fact that you're looking for an element whose `@class` attribute value *contains* the class name instead of one whose `@class` attribute value *is* the class name stems, most simply, from the fact that a specialized element's class contains both the class of its base element and its unique class. You can process the specialized element via an existing template by matching on the part of the class it shares with its base element, or you can process it uniquely in its own template by matching on the part of the class that is specific to the specialization. Don't get too hung up on this; just take away that the DITA OT templates match elements based on their class, not their name.

Line 15 starts `ditaval` processing for the `<codeblock>` element. Line 16 calls another template that processes `@spectitle`, if present. `<xsl:call-template>` is somewhat similar to `<xsl:apply-templates>` except



that `<xsl:call-template>` is more like calling a function whereas `<xsl:apply-templates>` is more for element processing.

Line 18 creates the XHTML pre element that will hold the content of the DITA `<codeblock>` element. Lines 19-21 call additional templates. The processing in these templates applies to many elements and they are called from multiple templates throughout the XHTML plugin.

Line 23 creates the XHTML code element that will hold the content of the DITA `<codeblock>` element. Line 24 applies the template; that is, it processes the content of the element selected by the template—`<codeblock>`.

Lines 25 and 26 close the XHTML `<code>` and `<pre>` elements, respectively. Line 28 ends ditaval processing for the `<codeblock>` element. Line 29 ends the "codeblock" template.

An example of a DITA to PDF template

Now let's look at the "codeblock" template in the PDF plugin. This template is found in the **pr-domain.xsl** file, which is in the `xsl\fo` sub-folder of the **org.dita.pdf2** plugin. There are a lot more stylesheets in the PDF plugin than in the XHTML plugin, and that's mainly due to a couple of things:

- The plugins were developed by different groups who made different decisions about what to group together (for example, the elements found in **dita2htmlImpl.xsl** in **org.dita.xhtml** are scattered amongst several stylesheets in **org.dita.pdf2**).
- There are additional PDF stylesheets for things like font processing and elements specific to print output (like prefaces, frontmatter, and page generation).

As you work with the various plugins, you'll become accustomed to where to find the templates you need.

```

52 <xsl:template match="*[contains(@class,' pr-d/codeblock ')]">
53   <xsl:call-template name="generateAttrLabel"/>
54   <fo:block xsl:use-attribute-sets="codeblock">
55     <xsl:call-template name="commonattributes"/>
56     <xsl:call-template name="setFrame"/>
57     <xsl:call-template name="setScale"/>
58     <xsl:call-template name="setExpand"/>
59     <xsl:variable name="codeblock.line-number" as="xs:boolean">
60       <xsl:apply-templates select="." mode="codeblock.generate-line-number"/>
61     </xsl:variable>
62     <xsl:choose>
63       <xsl:when test="$codeblock.wrap or $codeblock.line-number">
64         <xsl:variable name="content" as="node()*">
65           <xsl:apply-templates/>
66         </xsl:variable>
67         <xsl:choose>
68           <xsl:when test="$codeblock.line-number">
69             <xsl:variable name="buf" as="document-node()">
70               <xsl:document>
71                 <xsl:processing-instruction name="line-number"/>
72                 <xsl:apply-templates select="$content" mode="codeblock.line-number"/>
73               </xsl:document>
74             </xsl:variable>
75             <xsl:variable name="line-count"
76               select="count($buf/descendant::processing-instruction('line-number'))"/>
77             <xsl:apply-templates select="$buf" mode="codeblock">
78               <xsl:with-param name="line-count" select="$line-count" tunnel="yes"/>
79             </xsl:apply-templates>
80           </xsl:when>
81           <xsl:otherwise>
82             <xsl:apply-templates select="$content" mode="codeblock"/>
83           </xsl:otherwise>
84         </xsl:choose>
85       </xsl:when>
86       <xsl:otherwise>
87         <xsl:apply-templates/>
88       </xsl:otherwise>
89     </xsl:choose>
90   </fo:block>
</xsl:template>

```

Figure 6: Codeblock template (org.dita.pdf2 plugin)

This template is much longer than the XHTML template, mainly because it includes some line-numbering functionality that is not available in XHTML version. Let's dissect it.

Line 52 starts the template. Notice that the @match attribute is the same as the one in the XHTML template. Line 53 calls another template that processes @spectitle, if present. Line 54 begins the <fo:block> element that will hold the contents of the DITA <codeblock> element.

This last point is one of the significant differences between the PDF plugin and the other DITA OT plugins. As mentioned earlier, DITA XML has to be converted to a different XML tagset called XSL-FO so that it can be processed by PDF renderers. XSL-FO enables you to specify things that are outside the usual realm of HTML-based outputs such as page dimensions, headers, footers, and so on. The XSL-FO element that contains a block of text is <fo:block>. So instead of putting the content of codeblock into <pre> or <code> as for XHTML, you put it in <fo:block> for PDF outputs. Anytime you see an element that starts with “fo:”, you know it is an XSL-FO element. (“fo:” is a *namespace*, which is a way of signifying that an element belongs to a particular XML tagset. The “xsl:” prefix in front of all the other elements in the template is another namespace. For an overview of namespaces, check out https://www.w3schools.com/xml/xml_namespaces.asp).

Lines 55-58 call additional templates. The processing in these templates applies to many elements and they are called from multiple other templates throughout the XHTML plugin.

Lines 59-61 create a variable named “codeblock.line-number” for use later in the template. Line 60 is an <xsl:apply-templates> but with a twist. It includes a @mode attribute with a value of “codeblock.generate-line-number”. In this case, @mode means, essentially, “get everything ready to do something but don’t actually do it until I tell you to.”

Line 62 starts an <xsl:choose> element, which, as its name suggests, sets up several options for the template to follow based on some criteria such as a parameter, XML tree structure, attribute value, and so on. Within an <xsl:choose> element, you have one or more <xsl:when> elements that specify the case to test for and the action to take if the case is met, and an <xsl:otherwise> element which specifies what to do when none of the <xsl:when> criteria are met. Line 63 starts the first <xsl:when> element.

Lines 64-66 define another variable named “content.”

An <xsl:when> element can contain another <xsl:choose> element, as we see in line 67. Nested <xsl:choose> elements simply specify that when a set of <xsl:when> criteria is met, perform another set of tests.

Line 68 tests for the presence of a parameter named “codeblock.line-number.” For a PDF output, users can specify whether or not to include line numbers with codeblocks. If the user has specified yes, then all of the processing within the <xsl:when> element will take place. If the user has specified no, then only the processing in the <xsl:otherwise> element (lines 80-82) takes place. Lines 69-78 create the line numbers for the <codeblock>. Line 79 ends the first, nested, <xsl:when> element. As mentioned previously, lines 80-82 specify what happens if the user chooses not to have line numbers in the PDF output. Line 83 ends the nested <xsl:choose> element and line 84 ends the highest-level, un-nested <xsl:when> element.

Lines 85-87 outline what happens when the <xsl:when> test in line 63 is not true (that is, no line wrapping and no line numbers in the codeblock). Line 88 ends the parent <xsl:choose> element. Line 89 closes the <fo:block> element and line 90 ends the template.

What to do next

If you're not responsible for maintaining the stylesheets for your content output, you can simply relax and enjoy this new bit of high-level XSL knowledge. However, if you're curious to know more, you can continue to research XSL on the Web or you can roll up your sleeves and dig in a little bit. There are many, many online and print resources available for learning XML and XSL...and many of them are free!

Start simple. Create a simple XML file of your own (perhaps your own addressbook, or list of DVDs, or international capitals with their populations and elevations, or whatever interests you). Then create a basic stylesheet to transform them to HTML.

If you are using an XML editor that includes transformation capability then you don't have any extra setup to do for your practice. Saxon or Xalan and all the necessary associated files are already installed and ready to go.

After you've had a little hands-on experience, you can dive into working with DITA OT stylesheets. If you have a custom plugin, ask its owner if they can create a sandbox copy for you to play around with. Or create your own! The DITA OT documentation (in the [DITA-OT] \doc\dev_ref folder) explains how the DITA OT plugins work and how to create your own. Specifically, take a look at the [plugins-overview.html](#) topic. For a deep dive into PDF plugins, *DITA For Print: A DITA Open Toolkit Workbook*, by Leigh White (yours truly), available from [XML Press](#), offers a full explanation of how the **org.dita.pdf2** plugin works and provides step-by-step instructions for creating your own PDF plugin.

Good luck!

