Chapter 1. DocBook XSL

Bob Stayton

$Id: publishing.xml,v 1.4 2002/06/03 19:26:58 xmldoc Exp $

Table of Contents

Using XSL tools to publish DocBook documents 1
XSLT engines 1
A brief introduction to XSL 3
XSL processing model 5
Context is important 5
Programming features 6
Generating HTML output. 9
Generating formatting objects. 10
Customizing DocBook XSL stylesheets 10
Stylesheet inclusion vs. importing 11
Customizing with <xsl:import> 12
Setting stylesheet variables 12
Writing your own templates 13
Writing your own driver 13
Localization 14

Using XSL tools to publish DocBook documents

There is a growing list of tools to process DocBook documents using XSL stylesheets. Each tool implements parts or all of the XSL standard, which actually has several components:

Extensible Stylesheet Language (XSL)	A language for expressing stylesheets written in XML. It includes the formatting object language, but refers to separate documents for the transformation language and the path language.
XSL Transformation (XSLT)	The part of XSL for transforming XML documents into other XML documents, HTML, or text. It can be used to rearrange the content and generate new content.
XML Path Language (XPath)	A language for addressing parts of an XML document. It is used to find the parts of your document to apply different styles to. All XSL processors use this component.

To publish HTML from your XML documents, you just need an XSLT engine. To get to print, you need an XSLT engine to produce formatting objects (FO), which then must be processed with an FO engine to produce PostScript or PDF output.

XSLT engines

This section provides a discussion about which XSLT engines you might want to use to generate HTML and FO output from your DocBook XML documents, along with a few short examples of how to actually use some specific XSLT engines to generate that output. Before using any particular XSLT engine, you should consult its reference documentation for more detailed information.

Which XSLT engine should I use?

Currently, the only XSLT engines that are recommended and known to work well with the DocBook XSL stylesheets are Daniel Veillard's C-based implementation, xsltproc (the command line processor packaged with libxslt [http://xmlsoft.org/XSLT/], the XSLT C library for Gnome), and Michael Kay's Java-based implementation, Saxon [http://saxon.sourceforge.net/].

XSLT engines not recommended for use with DocBook

The following engines are not currently recommended for use with the DocBook XSL stylesheets:

James Clark's XT	XT is an incomplete implementation of the XSLT 1.0 specification. One of the important things that's missing from it is support for XSLT "keys", which the DocBook XSLT stylesheets rely on for generating indexes, among other things. So you can't use XT reliably with current versions of the stylesheets.
Xalan (both Java and C++ implementations)	Bugs in current versions of Xalan prevent it from being used reliably with the stylesheets.

Your choice of an XSLT engine may depend a lot on the environment you'll be running the engine in. Many DocBook users who need or want a non-Java application are using xsltproc. It's very fast, and also a good choice because Veillard monitors the DocBook mailing lists to field usage and troubleshooting questions and responds very quickly to bug reports. (And the libxslt site features a DocBook page [http://xmlsoft.org/XSLT/docbook.html] that, among other things, includes a shell script you can use to automatically generate XML catalogs [http://xmlsoft.org/catalog.html] for DocBook.) But one current limitation xsltproc has is that it doesn't yet support Norm Walsh's DocBook-specific XSLT extension functions.

If you can use a Java-based implementation, choose Michael Kay's Saxon. It supports Norm Walsh's DocBook-specific XSLT extension functions.

A variety of XSLT engines are available. Not all of them are used much in the DocBook community, but here's a list of some free/open-source ones you might consider (though xsltproc and Saxon are currently the only recommended XSLT engines for use with DocBook).

- xsltproc, written in C, from Daniel Veillard (http://xmlsoft.org/XSLT/)
- Saxon, written in Java, from Michael Kay (http://saxon.sourceforge.net/)
- 4XSLT, written in Python, from FourThought LLC (http://www.fourthought.com)
- Sablotron, written in C++, from Ginger Alliance (http://www.gingerall.com)
- XML::XSLT,written in Perl, from Geert Josten and Egon Willighagen (http://www.cpan.org)

For generating print/PDF output from FO files, there are two free/open-source FO engines that, while they aren't complete bug-free implementations of the FO part of the XSL specification, are still very useful:

- PassiveTeX (TeX-based) from Sebastian Rahtz (http://www.hcu.ox.ac.uk/TEI/Software/passivetex/)
- FOP (Java-based) from the Apache XML Project (http://xml.apache.org/fop/)

Of those, PassiveTeX currently seems to be the more mature, less buggy implementation.

And there are two proprietary commercial products that both seem to be fairly mature, complete implementations of the FO part of the XSL specification:

- current versions of Arbortext Epic Editor [http://www.arbortext.com] include integrated support for processing formatting object files
- RenderX XEP [http://www.renderx.com] (written in Java) is a standalone tool for processing formatting object files

How do I use an XSLT engine?

Before using any XSLT engine, you should consult the reference documentation that comes with it for details about its command syntax and so on. But there are some common steps to follow when using the Java-based engines, so here's an example of using Saxon from the UNIX command line that might help give you general idea of how to use the Java-based engines.

Note

You'll need to alter your *CLASSPATH* environment variable to include the path to where you put the `saxon.jar` file from the Saxon distribution. And you'll need to specify the correct path to the `docbook.xsl` HTML stylesheet file in your local environment.

Example 1.1. Using Saxon to generate HTML output

```
CLASSPATH=saxon.jar:$CLASSPATH
export CLASSPATH
java  com.icl.saxon.StyleSheet  filename.xml docbook/html/docbook.xsl > output.html
```

If you replace the path to the HTML stylesheet with the path to the FO stylesheet, Saxon will produce a formatting object file. Then you can convert that to PDF using a FO engine such such as FOP, the free/open-source FO engine available from the Apache XML Project (http://xml.apache.org/fop/). Here is an example of that two-stage process.

Example 1.2. Using Saxon and FOP to generate PDF output

```
CLASSPATH=saxon.jar:fop.jar:$CLASSPATH
export CLASSPATH
java  com.icl.saxon.StyleSheet filename.xml docbook/fo/docbook.xsl > output.fo
java  org.apache.fop.apps.CommandLine output.fo output.pdf
```

Using a C-based XSLT engine such as xsltproc is a little easier, since it doesn't require setting any environment variables or remembering Java package names. Here's an example of using xsltproc to generate HTML output.

Example 1.3. Using xsltproc to generate HTML output

```
xsltproc docbook/html/docbook.xsl filename.xml > output.html
```

Note that when using xsltproc, the pathname to the stylesheet file precedes the name of your XML source file on the command line (it's the other way around with Saxon and with most other Java-based XSLT engines).

A brief introduction to XSL

XSL is both a transformation language and a formatting language. The XSLT transformation part lets you scan through a document's structure and rearrange its content any way you like. You can write out the content using a different set of XML tags, and generate text as needed. For example, you can scan through a document to locate all headings and then insert a generated table of contents at the beginning of the document, at the same time writing out the content marked up as HTML. XSL is also a rich formatting language, letting you apply typesetting controls to all components of your output. With a good formatting backend, it is capable of producing high quality printed pages.

An XSL stylesheet is written using XML syntax, and is itself a well-formed XML document. That makes the basic syntax familiar, and enables an XML processor to check for basic syntax errors. The stylesheet instructions use special element names, which typically begin with `xsl:` to distinguish them from any XML tags you want to appear in the output. The XSL namespace is identified at the top of the stylesheet file. As with other XML, any XSL elements that are not empty will require a closing tag. And some XSL elements have specific attributes that control their behavior. It helps to keep a good XSL reference book handy.

Here is an example of a simple XSL stylesheet applied to a simple XML file to generate HTML output.

Example 1.4. Simple XML file

```
<?xml version="1.0"?>
<document>
<title>Using a mouse</title>
<para>It's easy to use a mouse. Just roll it
around and click the buttons.</para>
</document>
```

Example 1.5. Simple XSL stylesheet

```
<?xml version='1.0'?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version='1.0'>
<xsl:output method="html"/>

<xsl:template match="document">
<HTML><HEAD><TITLE>
<xsl:value-of select="./title"/>
</TITLE>
</HEAD>
<BODY>
<xsl:apply-templates/>
</BODY>
</HTML>
</xsl:template>

<xsl:template match="title">
<H1><xsl:apply-templates/></H1>
</xsl:template>

<xsl:template match="para">
<P><xsl:apply-templates/></P>
</xsl:template>

</xsl:stylesheet>
```

Example 1.6. HTML output

```
<HTML>
<HEAD>
<TITLE>Using a mouse</TITLE>
</HEAD>
<BODY>
<H1>Using a mouse</H1>
<P>It's easy to use a mouse. Just roll it
around and click the buttons.</P>
```

```
</BODY>
</HTML>
```

XSL processing model

XSL is a template language, not a procedural language. That means a stylesheet specifies a sample of the output, not a sequence of programming steps to generate it. A stylesheet consists of a mixture of output samples with instructions of what to put in each sample. Each bit of output sample and instructions is called a *template*.

In general, you write a template for each element type in your document. That lets you concentrate on handling just one element at a time, and keeps a stylesheet modular. The power of XSL comes from processing the templates recursively. That is, each template handles the processing of its own element, and then calls other templates to process its children, and so on. Since an XML document is always a single root element at the top level that contains all of the nested descendent elements, the XSL templates also start at the top and work their way down through the hierarchy of elements.

Take the DocBook `<para>` paragraph element as an example. To convert this to HTML, you want to wrap the paragraph content with the HTML tags `<p>` and `</p>`. But a DocBook `<para>` can contain any number of in-line DocBook elements marking up the text. Fortunately, you can let other templates take care of those elements, so your XSL template for `<para>` can be quite simple:

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

The `<xsl:template>` element starts a new template, and its `match` attribute indicates where to apply the template, in this case to any `<para>` elements. The template says to output a literal `<p>` string and then execute the `<xsl:apply-templates/>` instruction. This tells the XSL processor to look among all the templates in the stylesheet for any that should be applied to the content of the paragraph. If each template in the stylesheet includes an `<xsl:apply-templates/>` instruction, then all descendents will eventually be processed. When it is through recursively applying templates to the paragraph content, it outputs the `</p>` closing tag.

Context is important

Since you aren't writing a linear procedure to process your document, the context of where and how to apply each modular template is important. The `match` attribute of `<xsl:template>` provides that context for most templates. There is an entire expression language, XPath, for identifying what parts of your document should be handled by each template. The simplest context is just an element name, as in the example above. But you can also specify elements as children of other elements, elements with certain attribute values, the first or last elements in a sequence, and so on. Here is how the DocBook `<formalpara>` element is handled:

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

<xsl:template match="formalpara/title">
<b><xsl:apply-templates/></b>
<xsl:text> </xsl:text>
</xsl:template>

<xsl:template match="formalpara/para">
```

```
<xsl:apply-templates/>
</xsl:template>
```

There are three templates defined, one for the <formalpara> element itself, and one for each of its children elements. The match attribute value formalpara/title in the second template is an XPath expression indicating a <title> element that is an immediate child of a <formalpara> element. This distinguishes such titles from other <title> elements used in DocBook. XPath expressions are the key to controlling how your templates are applied.

In general, the XSL processor has internal rules that apply templates that are more specific before templates that are less specific. That lets you control the details, but also provides a fallback mechanism to a less specific template when you don't supply the full context for every combination of elements. This feature is illustrated by the third template, for formalpara/para. By including this template, the stylesheet processes a <para> within <formalpara> in a special way, in this case by not outputting the HTML <p> tags already output by its parent. If this template had not been included, then the processor would have fallen back to the template specified by match="para" described above, which would have output a second set of <p> tags.

You can also control template context with XSL *modes*, which are used extensively in the DocBook stylesheets. Modes let you process the same input more than once in different ways. A mode attribute in an <xsl:template> definition adds a specific mode name to that template. When the same mode name is used in <xsl:apply-templates/>, it acts as a filter to narrow the selection of templates to only those selected by the match expression *and* that have that mode name. This lets you define two different templates for the same element match that are applied under different contexts. For example, there are two templates defined for DocBook <listitem> elements:

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

<xsl:template match="listitem" mode="xref">
<xsl:number format="1"/>
</xsl:template>
```

The first template is for the normal list item context where you want to output the HTML <li> tags. The second template is called with <xsl:apply-templates select="$target" mode="xref"/> in the context of processing <xref> elements. In this case the select attribute locates the ID of the specific list item and the mode attribute selects the second template, whose effect is to output its item number when it is in an ordered list. Because there are many such special needs when processing <xref> elements, it is convenient to define a mode name xref to handle them all. Keep in mind that mode settings do *not* automatically get passed down to other templates through <xsl:apply-templates/>.

Programming features

Although XSL is template-driven, it also has some features of traditional programming languages. Here are some examples from the DocBook stylesheets.

```
Assign a value to a variable:
<xsl:variable name="refelem" select="name($target)"/>

If statement:
<xsl:if test="$show.comments">
<i><xsl:call-template name="inline.charseq"/></i>
</xsl:if>

Case statement:
<xsl:choose>
<xsl:when test="@columns">
<xsl:value-of select="@columns"/>
```

```
</xsl:when>
<xsl:otherwise>1</xsl:otherwise>
</xsl:choose>

Call a template by name like a subroutine, passing parameter values and accepting a return value:
<xsl:call-template name="xref.xreflabel">
<xsl:with-param name="target" select="$target"/>
</xsl:call-template>
```

However, you can't always use these constructs as you do in other programming languages. Variables in particular have very different behavior.

Using variables and parameters

XSL provides two elements that let you assign a value to a name: `<xsl:variable>` and `<xsl:param>`. These share the same name space and syntax for assigning names and values. Both can be referred to using the `$name` syntax. The main difference between these two elements is that a param's value acts as a default value that can be overridden when a template is called using a `<xsl:with-param>` element as in the last example above.

Here are two examples from DocBook:

```
<xsl:param name="cols">1</xsl:param>
<xsl:variable name="segnum" select="position()"/>
```

In both elements, the name of the parameter or variable is specified with the `name` attribute. So the name of the `param` here is `cols` and the name of the `variable` is `segnum`. The value of either can be supplied in two ways. The value of the first example is the text node "1" and is supplied as the content of the element. The value of the second example is supplied as the result of the expression in its `select` attribute, and the element itself has no content.

The feature of XSL variables that is odd to new users is that once you assign a value to a variable, you cannot assign a new value within the same scope. Doing so will generate an error. So variables are not used as dynamic storage bins they way they are in other languages. They hold a fixed value within their scope of application, and then disappear when the scope is exited. This feature is a result of the design of XSL, which is template-driven and not procedural. This means there is no definite order of processing, so you can't rely on the values of changing variables. To use variables in XSL, you need to understand how their scope is defined.

Variables defined outside of all templates are considered global variables, and they are readable within all templates. The value of a global variable is fixed, and its global value can't be altered from within any template. However, a template can create a local variable of the same name and give it a different value. That local value remains in effect only within the scope of the local variable.

Variables defined within a template remain in effect only within their permitted scope, which is defined as all following siblings and their descendants. To understand such a scope, you have to remember that XSL instructions are true XML elements that are embedded in an XML family hierarchy of XSL elements, often referred to as parents, children, siblings, ancestors and descendants. Taking the family analogy a step further, think of a variable assignment as a piece of advice that you are allowed to give to certain family members. You can give your advice only to your younger siblings (those that follow you) and their descendents. Your older siblings won't listen, neither will your parents or any of your ancestors. To stretch the analogy a bit, it is an error to try to give different advice under the same name to the same group of listeners (in other words, to redefine the variable). Keep in mind that this family is not the elements of your document, but just the XSL instructions in your stylesheet. To help you keep track of such scopes in hand-written stylesheets, it helps to indent nested XSL elements. Here is an edited snippet from the DocBook stylesheet file `pi.xsl` that illustrates different scopes for two variables:

```
1 <xsl:template name="dbhtml-attribute">
2 ...
```

```
3    <xsl:choose>
4       <xsl:when test="$count>count($pis)">
5          <!-- not found -->
6       </xsl:when>
7       <xsl:otherwise>
8          <xsl:variable name="pi">
9             <xsl:value-of select="$pis[$count]"/>
10         </xsl:variable>
11         <xsl:choose>
12            <xsl:when test="contains($pi,concat($attribute, '='))">
13               <xsl:variable name="rest" select="substring-after($pi,concat($attribute,'='))"/>
14               <xsl:variable name="quote" select="substring($rest,1,1)"/>
15               <xsl:value-of select="substring-before(substring($rest,2),$quote)"/>
16            </xsl:when>
17            <xsl:otherwise>
18            ...
19            </xsl:otherwise>
20         </xsl:choose>
21      </xsl:otherwise>
22   </xsl:choose>
23 </xsl:template>
```

The scope of the variable `pi` begins on line 8 where it is defined in this template, and ends on line 20 when its last sibling ends.[1] The scope of the variable `rest` begins on line 13 and ends on line 15. Fortunately, line 15 outputs an expression using the value before it goes out of scope.

What happens when an `<xsl:apply-templates/>` element is used within the scope of a local variable? Do the templates that are applied to the document children get the variable? The answer is no. The templates that are applied are not actually within the scope of the variable. They exist elsewhere in the stylesheet and are not following siblings or their descendants.

To pass a value to another template, you pass a parameter using the `<xsl:with-param>` element. This parameter passing is usually done with calls to a specific named template using `<xsl:call-template>`, although it works with `<xsl:apply-templates>` too. That's because the called template must be expecting the parameter by defining it using a `<xsl:param>` element with the same parameter name. Any passed parameters whose names are not defined in the called template are ignored.

Here is an example of parameter passing from `docbook.xsl`:

```
<xsl:call-template name="head.content">
<xsl:with-param name="node" select="$doc"/>
</xsl:call-template>
```

Here a template named `head.content` is being called and passed a parameter named `node` whose content is the value of the `$doc` variable in the current context. The top of that template looks like this:

```
<xsl:template name="head.content">
<xsl:param name="node" select="."/>
```

The template is expecting the parameter because it has a `<xsl:param>` defined with the same name. The value in this definition is the default value. This would be the parameter value used in the template if the template was called without passing that parameter.

[1] Technically, the scope extends to the end tag of the parent of the `<xsl:variable>` element. That is effectively the last sibling.

Generating HTML output.

You generate HTML from your DocBook XML files by applying the HTML version of the stylesheets. This is done by using the HTML driver file `docbook/html/docbook.xsl` as your stylesheet. That is the master stylesheet file that uses `<xsl:include>` to pull in the component files it needs to assemble a complete stylesheet for producing HTML.

The way the DocBook stylesheet generates HTML is to apply templates that output a mix of text content and HTML elements. Starting at the top level in the main file `docbook.xsl`:

```
<xsl:template match="/">
<xsl:variable name="doc" select="*[1]"/>
<html>
<head>
<xsl:call-template name="head.content">
<xsl:with-param name="node" select="$doc"/>
</xsl:call-template>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

This template matches the root element of your input document, and starts the process of recursively applying templates. It first defines a variable named `doc` and then outputs two literal HTML elements `<html>` and `<head>`. Then it calls a named template `head.content` to process the content of the HTML `<head>`, closes the `<head>` and starts the `<body>`. There it uses `<xsl:apply-templates/>` to recursively process the entire input document. Then it just closes out the HTML file.

Simple HTML elements can generated as literal elements as shown here. But if the HTML being output depends on the context, you need something more powerful to select the element name and possibly add attributes and their values. Here is a fragment from `sections.xsl` that shows how a heading tag is generated using the `<xsl:element>` and `<xsl:attribute>` elements:

```
1 <xsl:element name="h{$level}">
2   <xsl:attribute name="class">title</xsl:attribute>
3   <xsl:if test="$level<3">
4     <xsl:attribute name="style">clear: all</xsl:attribute>
5   </xsl:if>
6   <a>
7     <xsl:attribute name="name">
8       <xsl:call-template name="object.id"/>
9     </xsl:attribute>
10     <b><xsl:copy-of select="$title"/></b>
11   </a>
12 </xsl:element>
```

This whole example is generating a single HTML heading element. Line 1 begins the HTML element definition by identifying the name of the element. In this case, the name is an expression that includes the variable `$level` passed as a parameter to this template. Thus a single template can generate `<h1>`, `<h2>`, etc. depending on the context in which it is called. Line 2 defines a `class="title"` attribute that is added to this element. Lines 3 to 5 add a `style="clear all"` attribute, but only if the heading level is less than 3. Line 6 opens an `<a>` anchor element. Although this looks like a literal output string, it is actually modified by lines 7 to 9 that insert the `name` attribute into the `<a>` element. This illustrates that XSL is managing output elements as active element nodes, not just text strings. Line 10 outputs the text of the heading title, also passed as a parameter to the template, enclosed in HTML boldface tags.

Line 11 closes the anchor tag with the literal `</a>` syntax, while line 12 closes the heading tag by closing the element definition. Since the actual element name is a variable, it couldn't use the literal syntax.

As you follow the sequence of nested templates processing elements, you might be wondering how the ordinary text of your input document gets to the output. In the file `docbook.xsl` you will find this template that handles any text not processed by any other template:

```
<xsl:template match="text()">
<xsl:value-of select="."/>
</xsl:template>
```

This template's body consists of the "value" of the text node, which is just its text. In general, all XSL processors have some built-in templates to handle any content for which your stylesheet doesn't supply a matching template. This template serves the same function but appears explicitly in the stylesheet.

Generating formatting objects.

You generate formatting objects from your DocBook XML files by applying the fo version of the stylesheets. This is done by using the fo driver file `docbook/fo/docbook.xsl` as your stylesheet. That is the master stylesheet file that uses `<xsl:include>` to pull in the component files it needs to assemble a complete stylesheet for producing formatting objects. Generating a formatting objects file is only half the process of producing typeset output. You also need a formatting object processor such as the Apache XML Project's FOP as described in an earlier section.

The DocBook fo stylesheet works in a similar manner to the HTML stylesheet. Instead of outputting HTML tags, it outputs text marked up with `<fo:`*`something`*`>` tags. For example, to indicate that some text should be kept in-line and typeset with a monospace font, it might look like this:

```
<fo:inline-sequence font-family="monospace">/usr/man</fo:inline-sequence>
```

The templates in `docbook/fo/inline.xsl` that produce this output for a DocBook `<filename>` element look like this:

```
<xsl:template match="filename">
<xsl:call-template name="inline.monoseq"/>
</xsl:template>

<xsl:template name="inline.monoseq">
<xsl:param name="content">
<xsl:apply-templates/>
</xsl:param>
<fo:inline-sequence font-family="monospace">
<xsl:copy-of select="$content"/>
</fo:inline-sequence>
</xsl:template>
```

There are dozens of fo tags and attributes specified in the XSL standard. It is beyond the scope of this document to cover how all of them are used in the DocBook stylesheets. Fortunately, this is only an intermediate format that you probably won't have to deal with very much directly unless you are writing your own stylesheets.

Customizing DocBook XSL stylesheets

The DocBook XSL stylesheets are written in a modular fashion. Each of the HTML and FO stylesheets starts with a driver file that assembles a collection of component files into a complete stylesheet. This modular design puts similar things together into smaller files that are easier to write and maintain than one big stylesheet. The modular stylesheet files are distributed among four directories:

common/	contains code common to both stylesheets, including localization data
fo/	a stylesheet that produces XSL FO result trees
html/	a stylesheet that produces HTML/XHTML result trees
lib/	contains schema-independent functions

The driver files for each of HTML and FO stylesheets are `html/docbook.xsl` and `fo/docbook.xsl`, respectively. A driver file consists mostly of a bunch of `<xsl:include>` instructions to pull in the component templates, and then defines some top-level templates. For example:

```
<xsl:include href="../VERSION"/>
<xsl:include href="../lib/lib.xsl"/>
<xsl:include href="../common/l10n.xsl"/>
<xsl:include href="../common/common.xsl"/>
<xsl:include href="autotoc.xsl"/>
<xsl:include href="lists.xsl"/>
<xsl:include href="callout.xsl"/>
...
<xsl:include href="param.xsl"/>
<xsl:include href="pi.xsl"/>
```

The first four modules are shared with the FO stylesheet and are referenced using relative pathnames to the common directories. Then the long list of component stylesheets starts. Pathnames in include statements are always taken to be relative to the including file. Each included file must be a valid XSL stylesheet, which means its root element must be `<xsl:stylesheet>`.

Stylesheet inclusion vs. importing

XSL actually provides two inclusion mechanisms: `<xsl:include>` and `<xsl:import>`. Of the two, `<xsl:include>` is the simpler. It treats the included content as if it were actually typed into the file at that point, and doesn't give it any more or less precedence relative to the surrounding text. It is best used when assembling dissimilar templates that don't overlap what they match. The DocBook driver files use this instruction to assemble a set of modules into a stylesheet.

In contrast, `<xsl:import>` lets you manage the precedence of templates and variables. It is the preferred mode of customizing another stylesheet because it lets you override definitions in the distributed stylesheet with your own, without altering the distribution files at all. You simply import the whole stylesheet and add whatever changes you want.

The precedence rules for import are detailed and rigorously defined in the XSL standard. The basic rule is that any templates and variables in the importing stylesheet have precedence over equivalent templates and variables in the imported stylesheet. Think of the imported stylesheet elements as a fallback collection, to be used only if a match is not found in the current stylesheet. You can customize the templates you want to change in your stylesheet file, and let the imported stylesheet handle the rest.

Note

Customizing a DocBook XSL stylesheet is the opposite of customizing a DocBook DTD. When you customize a DocBook DTD, the rules of XML and SGML dictate that the *first* of any duplicate declarations wins. Any subsequent declarations of the same element or entity are ignored. The architecture of the DTD provides slots for inserting your own custom declarations early enough in the DTD for them to override the standard declarations. In contrast, customizing an XSL stylesheet is simpler because your definitions have precedence over imported ones.

You can carry modularization to deeper levels because module files can also include or import other modules. You'll need to be careful to maintain the precedence that you want as the modules get rolled up into a complete stylesheet.

Customizing with <xsl:import>

There is currently one example of customizing with <xsl:import> in the HTML version of the DocBook stylesheets. The xtchunk.xsl stylesheet modifies the HTML processing to output many smaller HTML files rather than a single large file per input document. It uses XSL extensions defined only in the XSL processor **XT**. In the driver file xtchunk.xsl, the first instruction is <xsl:import href="docbook.xsl"/>. That instruction imports the original driver file, which in turn uses many <xsl:include> instructions to include all the modules. That single import instruction gives the new stylesheet the complete set of DocBook templates to start with.

After the import, xtchunk.xsl redefines some of the templates and adds some new ones. Here is one example of a redefined template:

Original template in autotoc.xsl

```
<xsl:template name="href.target">
<xsl:param name="object" select="."/>
<xsl:text>#</xsl:text>
<xsl:call-template name="object.id">
<xsl:with-param name="object" select="$object"/>
</xsl:call-template>
</xsl:template>
```

New template in xtchunk.xsl

```
<xsl:template name="href.target">
<xsl:param name="object" select="."/>
<xsl:variable name="ischunk">
<xsl:call-template name="chunk">
<xsl:with-param name="node" select="$object"/>
</xsl:call-template>
</xsl:variable>

<xsl:apply-templates mode="chunk-filename" select="$object"/>

<xsl:if test="$ischunk='0'">
<xsl:text>#</xsl:text>
<xsl:call-template name="object.id">
<xsl:with-param name="object" select="$object"/>
</xsl:call-template>
</xsl:if>
</xsl:template>
```

The new template handles the more complex processing of HREFs when the output is split into many HTML files. Where the old template could simply output *#object.id*, the new one outputs *filename#object.id*.

Setting stylesheet variables

You may not have to define any new templates, however. The DocBook stylesheets are parameterized using XSL variables rather than hard-coded values for many of the formatting features. Since the <xsl:import> mechanism also lets you redefine global variables, this gives you an easy way to customize many features of the DocBook stylesheets. Over time, more features will be parameterized to permit customization. If you find hardcoded values in the stylesheets that would be useful to customize, please let the maintainer know.

Near the end of the list of includes in the main DocBook driver file is the instruction <xsl:include href="param.xsl"/>. The param.xsl file is the most important module for customizing a DocBook XSL stylesheet.

This module contains no templates, only definitions of stylesheet variables. Since these variables are defined outside of any template, they are global variables and apply to the entire stylesheet. By redefining these variables in an importing stylesheet, you can change the behavior of the stylesheet.

To create a customized DocBook stylesheet, you simply create a new stylesheet file such as `mystyle.xsl` that imports the standard stylesheet and adds your own new variable definitions. Here is an example of a complete custom stylesheet that changes the depth of sections listed in the table of contents from two to three:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version='1.0'
xmlns="http://www.w3.org/TR/xhtml1/transitional"
exclude-result-prefixes="#default">

<xsl:import href="docbook.xsl"/>

<xsl:variable name="toc.section.depth">3</xsl:variable>
<!-- Add other variable definitions here -->

</xsl:stylesheet>
```

Following the opening stylesheet element are the import instruction and one variable definition. The variable `toc.section.depth` was defined in `param.xsl` with value "2", and here it is defined as "3". Since the importing stylesheet takes precedence, this new value is used. Thus documents processed with `mystyle.xsl` instead of `docbook.xsl` will have three levels of sections in the tables of contents, and all other processing will be the same.

Use the list of variables in `param.xsl` as your guide for creating a custom stylesheet. If the changes you want are controlled by a variable there, then customizing is easy.

Writing your own templates

If the changes you want are more extensive than what is supported by variables, you can write new templates. You can put your new templates directly in your importing stylesheet, or you can modularize your importing stylesheet as well. You can write your own stylesheet module containing a collection of templates for processing lists, for example, and put them in a file named `mylists.xsl`. Then your importing stylesheet can pull in your list templates with a `<xsl:include href="mylists.xsl"/>` instruction. Since your included template definitions appear after the main import instruction, your templates will take precedence.

You'll need to make sure your new templates are compatible with the remaining modules, which means:

- Any named templates should use the same name so calling templates in other modules can find them.
- Your template set should process the same elements matched by templates in the original module, to ensure complete coverage.
- Include the same set of `<xsl:param>` elements in each template to interface properly with any calling templates, although you can set different values for your parameters.
- Any templates that are used like subroutines to return a value should return the same data type.

Writing your own driver

Another approach to customizing the stylesheets is to write your own driver file. Instead of using `<xsl:import href="docbook.xsl"/>`, you copy that file to a new name and rewrite any of the `<xsl:include/>` instructions to assemble a custom collection of stylesheet modules. One reason to do this is to speed up processing by reducing the

size of the stylesheet. If you are using a customized DocBook DTD that omits many elements you never use, you might be able to omit those modules of the stylesheet.

Localization

The DocBook stylesheets include features for localizing generated text, that is, printing any generated text in a language other than the default English. In general, the stylesheets will switch to the language identified by a `lang` attribute when processing elements in your documents. If your documents use the `lang` attribute, then you don't need to customize the stylesheets at all for localization.

As far as the stylesheets go, a `lang` attribute is inherited by the descendents of a document element. The stylesheet searches for a `lang` attribute using this XPath expression:

```
<xsl:variable name="lang-attr"
select="($target/ancestor-or-self::*/@lang
|$target/ancestor-or-self::*/@xml:lang)[last()]"/>
```

This locates the attribute on the current element or its most recent ancestor. Thus a `lang` attribute is in effect for an element and all of its descendents, unless it is reset in one of those descendents. If you define it in only your document root element, then it applies to the whole document:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.0//EN" "docbook.dtd">
<book lang="fr">
...
</book>
```

When text is being generated, the stylesheet checks the most recent `lang` attribute and looks up the generated text strings for that language in a localization XML file. These are located in the `common` directory of the stylesheets, one file per language. Here is the top of the file `fr.xml`:

```
<localization language="fr">

<gentext key="abstract"                 text="R&#x00E9;sum&#x00E9;"/>
<gentext key="answer"                   text="R:"/>
<gentext key="appendix"                 text="Annexe"/>
<gentext key="article"                  text="Article"/>
<gentext key="bibliography"             text="Bibliographie"/>
...
```

The stylesheet templates use the gentext key names, and then the stylesheet looks up the associated text value when the document is processed with that lang setting. The file `l10n.xml` (note the `.xml` suffix) lists the filenames of all the supported languages.

You can also create a custom stylesheet that sets the language. That might be useful if your documents don't make appropriate use of the `lang` attribute. The module `l10n.xsl` defines two global variables that can be overridden with an importing stylesheet as described above. Here are their default definitions:

```
<xsl:variable name="l10n.gentext.language"></xsl:variable>
<xsl:variable name="l10n.gentext.default.language">en</xsl:variable>
```

The first one sets the language for all elements, regardless of an element's `lang` attribute value. The second just sets a default language for any elements that haven't got a `lang` setting of their own (or their ancestors).