Wrox Programmer to Programmer™

Beginning XSLT and Xpath

Transforming XML Documents and Data

Ian Williams

wrox™

Updates, source code, and Wrox technical support at **www.wrox.com**

Beginning XSLT and XPath

Beginning
XSLT and XPath

Transforming XML Documents and Data

Ian Williams

Wiley Publishing, Inc.

Beginning XSLT and XPath: Transforming XML Documents and Data

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-47725-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2009929458

About the Author

Ian Williams is an information designer specializing in XML technologies, and a software technical writer. He worked in the U.K. publishing industry before getting involved in information technology at OWL International, developers of the one of the first commercial hypertext products. Ian was a product manager there, and later a consultant working with large corporate customers.

Since 1998 Ian has worked on technical writing and information-design projects, most recently for Nokia, Reuters, and Volantis. He is co-author with Pierre Greborio of *Professional InfoPath 2003*, also from Wrox Press.

Ian lives with his wife, Helen, in Kent, in a converted lifeboat station overlooking the English Channel.

Credits

Executive Editor
Carol Long

Development Editor
Tom Dinse

Technical Editor
Dan Squier

Production Editor
Eric Charbonneau

Copy Editor
Luann Rouff

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Proofreader
Candace English

Indexer
Johnna VanHoose Dinse

Contents

Introduction

Welcome to XSLT and XPath, two members of the W3C XML family of standards. This book concentrates on using XSLT and XPath to solve problems that you are likely to encounter every day in writing XSLT stylesheets. I have tried to focus most attention on the features that you will need frequently, while still treating other aspects of the subject in brief. You can find additional detailed information in the Quick Reference appendixes, and in more advanced works such as Michael Kay's *XSLT 2.0 and XPath 2.0 Programmers Reference*, also in the Wrox list.

Who This Book Is For

I assume that you have a sound knowledge of XML and related web standards, such as XML Schema and XHTML. In an introductory book like this, there isn't enough space to fill in background information on these subjects.

Conversely, I don't assume that you are familiar with a particular programming language, or that you necessarily have a strong programming background. The chapters include a few comparisons with other languages, and as you'll see, XSLT takes a different approach from most of them.

If you are an experienced web author, or a technical writer who works regularly with XML, there is no reason why you can't pick up XSLT, leveraging your existing knowledge and skills. Quite a few practitioners that I know come from this kind of background.

This book aims to give you a good grounding in the basics of XSLT and XPath, concentrating on version 2.0 of both standards. It is *definitely not* aimed at experienced XSLT 1.0 programmers who require a skills update. Explaining the often-very-detailed differences between the two versions would simply confuse matters for beginners, who will do better to learn how to use techniques that are appropriate to the latest version.

XSLT in Outline

According to the W3C specification, XSLT (Extensible Stylesheet Language: Transformations) is a ''language for transforming XML documents into other XML documents.'' Given the widespread use of XML for exchanging data between applications and its success as a means of creating a wide range of document types, it is now easy to see why such a language is useful.

XSLT 1.0 was published as a recommendation late in 1999 shortly after XML 1.0, and at the time it didn't seem obvious that XSLT would become a success. It is probably so successful because it was, and still

is, most often used to generate HTML content for the web from XML sources — so much so that XSLT processing was incorporated into browser engines at an early date. XSLT 2.0 was published in January 2007 after a very long development period, which was complicated by some controversy and the need to track the development of XPath 2.0, on which is relies heavily.

> *The name Extensible Stylesheet Language: Transformations suggests that there is another ''branch'' to XSL — and there is: Extensible Stylesheet Language: Formatting Objects, or XSL-FO. XSL was initially part of a much more comprehensive project, covering both transformation and formatting semantics. The Formatting Objects recommendation (still formally titled ''XSL'') was published separately in October 2001. It is essentially an XML vocabulary used to specify the layout and properties of parts of printed pages, but I won't be covering it in this book.*

XSLT Is Different

Writing code to handle XML transformations in XSLT differs markedly from the approach used in other programming languages. It is written in XML syntax in a declarative fashion, with processing specified in pattern-matching rules. By *declarative* I mean the opposite of the usual imperative approach; that is, an XSLT programmer does not define a sequence of actions, but specifies a number of rules that the result should satisfy.

XSLT has a type system based on XML Schema, and XPath expressions form an important second language, matching source document objects, selecting content for processing, and performing operations on content.

Compared to some other languages, it is much easier to learn the XSLT basics, but the different syntax and the nature of the XSLT processing model take some getting used to.

The XML source example that follows is written in the DocBook vocabulary (which is widely used in documenting information technology):

```
<?xml version="1.0" encoding="UTF-8"?>
<article>

   <title>A Simple Transform</title>

   <para>Because the transform is an XML document we need to start with an XML
      declaration, specifying the version and the encoding.</para>
   <para>The root element in a stylesheet is
      <emphasis>xsl:stylesheet</emphasis>, though the synonym
         <emphasis>xsl:transform</emphasis> may also be used. You must always
      specify the XSLT namespace, and it is important to set the version
      attribute correctly to match the type of processing required. In this book
      we generally specify version 2.0.</para>

   <programlisting><![CDATA[
<xml version="1.0" encoding="UTF-8"/>
<xsl:stylesheet
```

```
      xmlns:xsl=http://www.w3.org/1999/XSL/Transform
      version="2.0"
      ...
   </xsl:stylesheet>]]></programlisting>

      <para>In both cases there are stylesheets available for creating XHTML and
         PDF output. But what if you need to migrate content from one system to
         another?</para
       ...
   </article>
```

Shown next is a simple XSLT stylesheet that transforms the XML to HTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>
               <xsl:value-of select="title"/>
            </title>
         </head>
         <body>
            <xsl:apply-templates select="body"/>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="title">
      <h1><xsl:value-of select="."/></h1>
   </xsl:template>
   <xsl:template match="para">
      <p><xsl:apply-templates/></p>
   </xsl:template>
   <xsl:template match="emphasis">
      <em><xsl:value-of select="."/></em>
   </xsl:template>
   <xsl:template match="programlisting">
      <pre>
         <xsl:value-of select="."/>
      </pre>
   </xsl:template>
</xsl:stylesheet>
```

The core HTML elements, highlighted in the preceding code example, form the output structure. They are written literally inside a series of sibling template instructions prefixed with `xsl:`.

The use of templates is similar to the non-HTML code you may have seen in web pages written using ASP, JSP, or PHP; but here the syntax is entirely XML, and the XSLT elements provide both the framework and the processing instructions.

Importance of XPath Language

XPath 2.0 is an expression language that is absolutely fundamental to XSLT 2.0 in several ways. An expression takes one or more input values and returns a value as output, so everywhere you can use a value you should also be able to use an expression to be evaluated. Usually, expressions are used as the value of attributes on XSLT elements — for example, `<xsl:value-of select="a+b"/>`.

XPath is used to match elements to XSLT template rules.

Another common use of XPath is selecting nodes in an XML document for subsequent processing. You can make a document-wide selection and refer to all the `<list>` elements, or be very specific by pointing to the `class` attribute in the first `<para>` in the third `<section>` of a document.

Then, you can use XPath to load documents, search strings, and manipulate numbers, using a very wide range of built-in functions.

Node Trees

XPath expressions operate on an abstract tree structure. Objects in the tree are nodes, of which there seven types, briefly described here:

- **Document:** The root of the tree representing an entire source document. I use the term *document node* to designate this node, to avoid confusing it with the root element of the source document.
- **Element:** Defined by pairs of start and end tags (e.g., `<title></title>`) or an empty element tag such as `<img/>` with no content.
- **Text:** A character sequence in an element.
- **Attribute:** The name and value of an attribute in an element start tag or an empty element tag, including all default value attributes in the schema.
- **Comment:** Comments in the XML source document, i.e., `<!-- -->`.
- **Processing instruction:** An instruction in the source document contained by `<? ?>`.
- **Namespace:** Namespace declaration copied to each element to which the declaration applies.

Figure I-1 shows how part of the tree of nodes for the DocBook article would look, with the document node outside of everything. Only the document, element, and text nodes for one instance of each element are shown. Each node contains the node type at the top, its name in the center in the case of elements, and the string value in the case of text nodes.

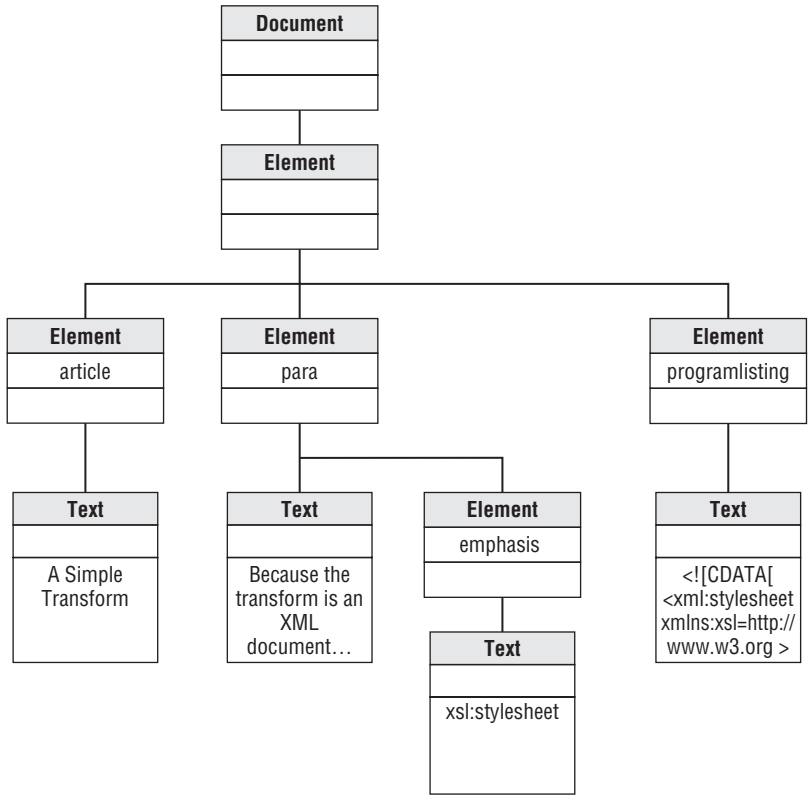

Figure I-1

Processing Overview

The basic work of an XSLT processor is to use a stylesheet as a set of instructions for producing a result document from a source document. Generally, all three documents are XML documents, so XSLT is said to *transform* one input object to an output object of the same kind. Figure I-2 illustrates the process in outline.

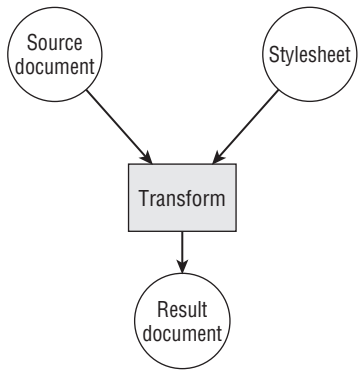

Figure I-2

An XSLT processor treats the input and output documents as trees of nodes. You can think of these trees as being something like the W3C document object model, which some XSLT processors indeed use. However, unlike the DOM, there is no defined API in the XSLT specification. Different processors are free to implement this abstract data model in different ways.

The basic processing sequence comprises several steps, shown in Figure I-3.

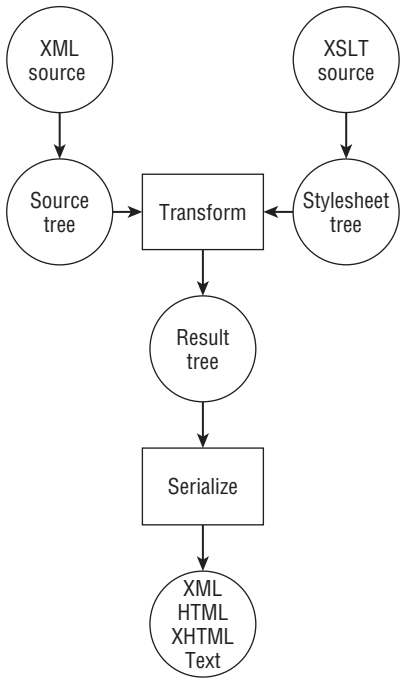

Figure I-3

1. The XML source document is parsed into a source tree.
2. The XSLT stylesheet is parsed to a stylesheet tree.
3. A transform is applied to create a result tree.
4. Serialization is applied to deliver content in the specified output format.

Essentially, the processor traverses the source tree in document order and looks for matching template rules in the stylesheet. If a match is found, then the instructions in the template are used to construct a node in the result tree. By default, the serialization creates an XML document, but specific instructions may be applied to output HTML, XHTML, or plain text.

The process can be more complex, potentially involving multiple sources, stylesheets and result trees, temporary trees held as variables, and multiple serializations. In Appendix B you'll find a more detailed view of how an XSLT processor works.

About the XSLT 2.0 Schema

As you work through the examples in the book, I'll introduce parts of the XSLT 2.0 schema so that you can examine the structure of the individual elements.

This schema is published separately by W3C, and it is not part of the XSLT 2.0 recommendation as such. The full schema is reproduced in Appendix E, and the latest version is also at `www.w3.org/2007/schema-for-xslt20.xsd`.

XSLT elements are broadly divided into two categories: declarations and instructions. For clarity, I mostly use one or the other of these terms, rather than call them elements.

The `<xsl:declaration>` and `<xsl:instruction>` are represented in the schema as abstract elements, which *never* appear in document instances, so you will not use them in a stylesheet; rather, you will use one of the elements in their substitution groups. A substitution group determines where the elements may appear. For example, you can see that `<xsl:output>` is a declaration from the `substitutionGroup` attribute value:

```
<xs:element name="output" substitutionGroup="xsl:declaration">
...
</xs:element>
```

This leads to a rather flat structure overall, with very little nesting of elements.

Declarations

Declarations define values such as the location of stylesheets to include or import, the method of output, global parameters, and the templates to use to match the source XML. These are top-level elements that immediately follow the root `<xsl:stylesheet>` element. They can appear in any order unless there is an `<xsl:import>` element, which must always appear first.

The schema declares the complex type `xsl:generic-element-type` with some common attributes:

```
<xs:complexType name="generic-element-type" mixed="true">
   <xs:attribute name="default-collation" type="xsl:uri-list"/>
   <xs:attribute name="exclude-result-prefixes" type="xsl:prefix-list-or-all"/>
   <xs:attribute name="extension-element-prefixes" type="xsl:prefix-list"/>
   <xs:attribute name="use-when" type="xsl:expression"/>
   <xs:attribute name="xpath-default-namespace" type="xs:anyURI"/>
   <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
```

`<xsl:declaration>` is then defined as a `generic-element-type`, and the top-level elements are subsequently specified to be in the declaration substitution group:

```
<xs:element name="declaration" type="xsl:generic-element-type" abstract="true"/>
```

Here, for example, is the schema definition for the <xsl:output> element:

```
<xs:element name="output" substitutionGroup="xsl:declaration">
   <xs:complexType mixed="true">
      <xs:complexContent mixed="true">
         <xs:extension base="xsl:generic-element-type">
            <xs:attribute name="name" type="xsl:QName"/>
            <xs:attribute name="method" type="xsl:method"/>
            <xs:attribute name="byte-order-mark" type="xsl:yes-or-no"/>
            <xs:attribute name="cdata-section-elements" type="xsl:QNames"/>
            <xs:attribute name="doctype-public" type="xs:string"/>
            <xs:attribute name="doctype-system" type="xs:string"/>
            <xs:attribute name="encoding" type="xs:string"/>
            <xs:attribute name="escape-uri-attributes" type="xsl:yes-or-no"/>
            <xs:attribute name="include-content-type" type="xsl:yes-or-no"/>
            <xs:attribute name="indent" type="xsl:yes-or-no"/>
            <xs:attribute name="media-type" type="xs:string"/>
            <xs:attribute name="normalization-form" type="xs:NMTOKEN"/>
            <xs:attribute name="omit-xml-declaration" type="xsl:yes-or-no"/>
            <xs:attribute name="standalone" type="xsl:yes-or-no-or-omit"/>
            <xs:attribute name="undeclare-prefixes" type="xsl:yes-or-no"/>
            <xs:attribute name="use-character-maps" type="xsl:QNames"/>
            <xs:attribute name="version" type="xs:NMTOKEN"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

Figure I-4 shows a schema diagram for the declaration substitutions.

Instructions

Other XSLT elements known as *instructions* are used to specify the construction of result trees from individual elements and attributes in the source XML.

The xsl:versioned-element-type is defined as an extension of the generic-element-type, and followed by the instruction declaration. This is because every element except <xsl:output> may have a version attribute containing the XSLT version number, which may be used to indicate which version of XSLT the processor should apply:

```
<xs:complexType name="versioned-element-type" mixed="true">
  <xs:complexContent>
    <xs:extension base="xsl:generic-element-type">
      <xs:attribute name="version" type="xs:decimal" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="instruction" type="xsl:versioned-element-type" abstract="true"/>
```

<xsl:output> *has an attribute with the same name, but this is intended to refer to the XML version specified in the output method.*

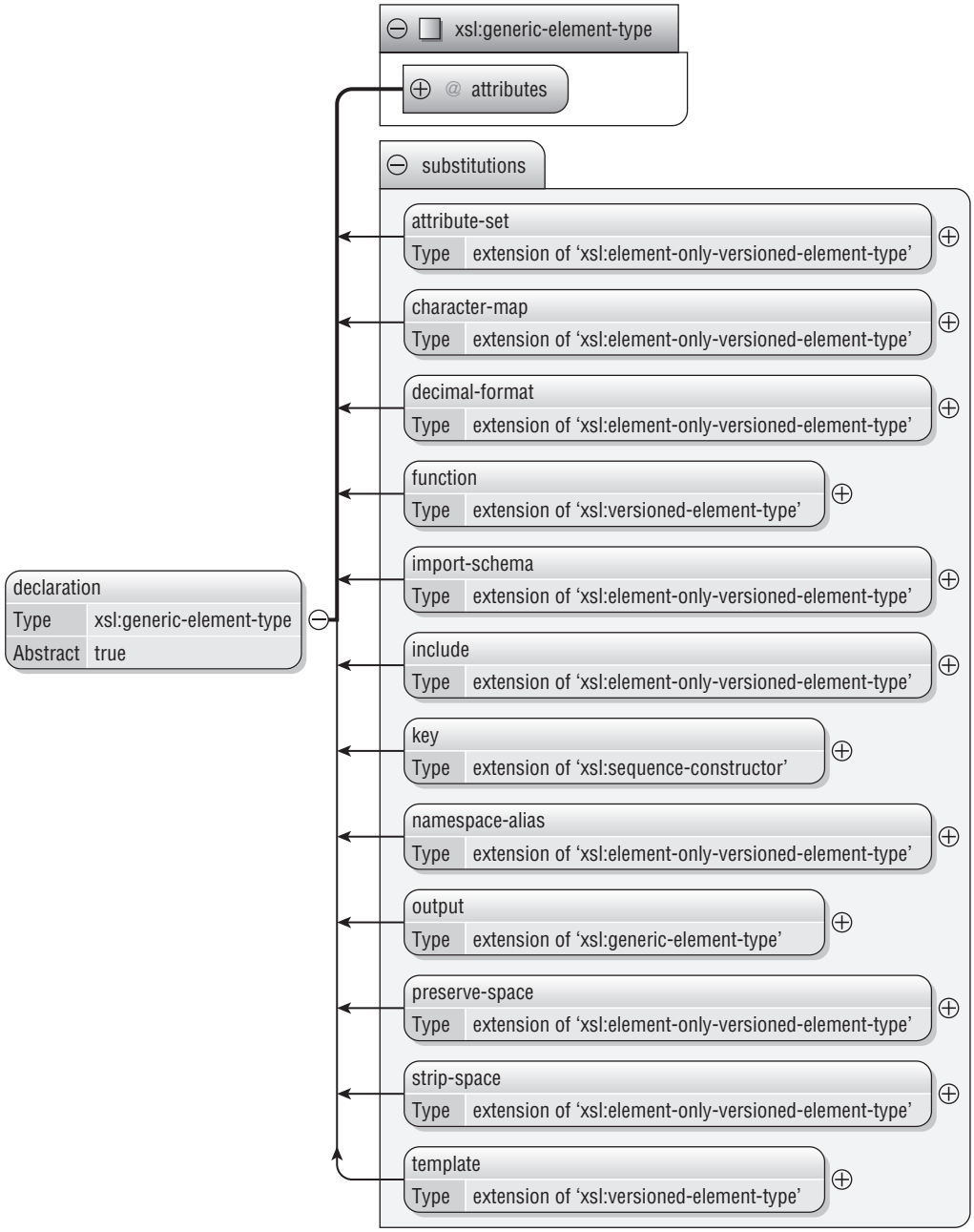

Figure I-4

`<xsl:value-of>`, which you will meet in Chapter 1, is specified as an instruction. It is also has the type *sequence constructor*, which is a type that contains a series of XSLT instructions. The

`xsl:sequence-constructor` type includes elements that can contain a sequence constructor. The schema extends the `xsl:versioned-element-type`, specifies the content model of a `sequence-constructor` group, and defines the `<xsl:value-of>` instruction:

```
<xs:complexType name="sequence-constructor">
  <xs:complexContent mixed="true">
    <xs:extension base="xsl:versioned-element-type">
      <xs:group ref="xsl:sequence-constructor-group" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:group name="sequence-constructor-group">
  <xs:choice>
    <xs:element ref="xsl:variable"/>
    <xs:element ref="xsl:instruction"/>
    <xs:group ref="xsl:result-elements"/>
  </xs:choice>
</xs:group>

<xs:element name="value-of" substitutionGroup="xsl:instruction">

  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">

        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="disable-output-escaping" type="xsl:yes-or-no"
          default="no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

The substitution diagram for instructions looks very similar to the one for declarations, but because of the number of instructions it is too large to include here.

What You Need to Use This Book

I habitually use a limited set of tools and a single development environment: the open-source Eclipse IDE, and the edition of Oxygen XML Editor that goes with it. Now and then I'll refer to them in particular.

There is a wide and very useful range of XSLT processors and XML editors out there; and while I don't want to endorse one rather than another, there are some arguments for using Oxygen as you work through this book, even if only temporarily:

- Oxygen is a multiplatform Java application.
- Both the basic and schema-aware versions of the Saxon XSLT processor are bundled with it.