XSLT

XML documents are especially useful when they're transformed into familiar formats, such as web pages, PDF files, or Java code—and those conversions require mastery of XSLT. This practical book not only teaches you how to be productive using XSLT, it also serves as a handy, dictionary-style reference to all the features and functions of the language that you'll need on the job. With XSLT, Second Edition, you will:

- Learn XSLT basics, including simple stylesheets and methods for setting up transformation engines
- Walk through the many parts of XSLT, particularly the template-based approach to transformations
- Understand the basics of XPath versions 1.0 and 2.0—the language used to describe parts of an XML document
- See how XML Schema support works in XSLT 2.0, including how to define elements and datatypes and use them in your stylesheets
- Get hundreds of stylesheets, including examples for every element, function, and operator defined by XSLT and XPath
- Use examples that apply both XSLT 1.0 and 2.0 solutions to the same problems so you can decide which version is more appropriate for your project

This new edition includes a complete set of stylesheet examples for both XSLT 1.0 and 2.0. You also get a thorough explanation of the relationship between XSLT, XPath, and other web standards. If you use XSLT, this book is the one resource you'll want on hand to help you solve problems quickly and accurately.

“The best review I received for the first edition of this book began, ‘I will never read this book.’ This was actually a positive review, as the reviewer went on to explain. ‘When I have a problem, I grab this book off the shelf, go to the index, and within five minutes I’ve found the answer to my problem. Then I toss it back on the shelf.’”

—Doug Tidwell, from the Preface of this book

Doug Tidwell, a senior programmer at IBM, has been working with markup languages for more than two decades. An expert who has been involved with XML since he spoke at the first SGML/XML conference in 1997, he teaches XML classes around the world.
XSLT
XSLT, Second Edition
by Doug Tidwell

Copyright © 2008 O’Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Simon St. Laurent
Production Editor: Sarah Schneider
Proofreader: Mary Brady
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

Nutshell Handbook, the Nutshell Handbook logo, and the O’Reilly logo are registered trademarks of O’Reilly Media, Inc. XSLT, the image of a Jabiru, and related trade dress are trademarks of O’Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O’Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-52721-1
To my family—my wonderful wife, Sheri Castle, and our amazing daughter, Lily—for their love, support, and understanding. Nothing I do would be possible or meaningful without them.

...and a special thanks to our dog, Domino, who frequently and selflessly pushed his fuzzy head between my hands and keyboard to protect me from carpal tunnel syndrome. Good boy!
# Table of Contents

**Preface** .................................................. xi

1. **Getting Started** ........................................... 1
   The Design of XSLT ........................................... 1
   XML Basics ................................................. 4
   Installing XSLT Processors ................................ 20
   Summary .................................................... 24

2. **The Obligatory Hello World Example** ..................... 25
   Goals of This Chapter ....................................... 25
   Transforming Hello World .................................. 25
   How a Stylesheet Is Processed .............................. 27
   Stylesheet Structure ........................................ 30
   Sample Gallery ............................................. 36
   Summary ..................................................... 44

3. **XPath: A Syntax for Describing Needles and Haystacks** ... 45
   The XPath Data Model ....................................... 46
   Location Paths ............................................. 55
   Attribute Value Templates ................................ 66
   Datatypes ................................................... 67
   XPath Operators ............................................ 71
   [2.0] Comments in XPath Expressions ...................... 102
   [2.0] Types of XSLT 2.0 Processors ....................... 104
   The XPath View of an XML Document ..................... 104
   Summary ..................................................... 112

4. **Creating Output** ........................................... 113
   Goals of This Chapter ....................................... 113
   Generating Text ............................................ 113
   Numbering Things ......................................... 118
   Formatting Decimal Numbers .............................. 127
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Custom Collations</td>
<td>287</td>
</tr>
<tr>
<td>Generating Hidden Word Graphics</td>
<td>293</td>
</tr>
<tr>
<td>Example: Generating an SVG Pie Chart</td>
<td>303</td>
</tr>
<tr>
<td>Writing Extensions in Other Languages</td>
<td>326</td>
</tr>
<tr>
<td>Using Extension Functions from the EXSLT Library</td>
<td>330</td>
</tr>
<tr>
<td>Accessing a Database with an Extension Element</td>
<td>333</td>
</tr>
<tr>
<td>Creating a Photo Album with an Extension Element</td>
<td>339</td>
</tr>
<tr>
<td>Summary</td>
<td>360</td>
</tr>
<tr>
<td>A. XSLT Reference</td>
<td>361</td>
</tr>
<tr>
<td>B. XPath Reference</td>
<td>545</td>
</tr>
<tr>
<td>C. XSLT, XPath, and XQuery Function Reference</td>
<td>563</td>
</tr>
<tr>
<td>D. XML Schema Overview</td>
<td>871</td>
</tr>
<tr>
<td>E. [2.0] Regular Expressions</td>
<td>897</td>
</tr>
<tr>
<td>F. XSLT Formatting Codes</td>
<td>919</td>
</tr>
<tr>
<td>G. XSLT 2.0 Migration Guide</td>
<td>925</td>
</tr>
<tr>
<td>Glossary</td>
<td>933</td>
</tr>
<tr>
<td>Index</td>
<td>943</td>
</tr>
</tbody>
</table>
About This Book

The goal of this book is to help you make the most of XSLT, the Extensible Stylesheet Language for Transformations. It covers both XSLT 1.0 and XSLT 2.0, along with versions 1.0 and 2.0 of XPath, the XML Path Language. The two languages are designed to work together: XPath identifies the parts of an XML document that should be transformed, and XSLT says how the transformation should be done.

The first few chapters of the book cover the features of XSLT by solving common problems using the language. Once you’ve mastered those techniques, the last section of the book contains a complete set of examples for all the features of XSLT and XPath. The book is designed as a tutorial for learning the language as you’re getting started. Once you’re comfortable with XSLT, the book can be used as a dictionary-style reference for the features and functions of the language.

Where I’m Coming From

Before we begin, it’s only fair that I tell you my biases.

I Believe in Open, Platform-Neutral, Standards-Based Computing

If any part of your business life ties you down to anything closed, proprietary, or platform-specific, I encourage you to make some changes. This book shows you how to take charge of your data and move it from one place to another on your terms, and not your software vendor’s. XML is shifting the balance of power from vendors to software users. If your tools force you to work in unnatural ways or refuse to let you have your data when and where you want it, you don’t have to take it anymore.

I Assume You’re Busy

The best review I received of the first edition of this book began, “I will never read this book.” This was actually a positive review. As the reviewer went on to explain, “When
I have a problem, I grab this book off the shelf, go to the index, and within five minutes I’ve found the answer to my problem. Then I toss it back on the shelf.”

That’s exactly the kind of book I’ve tried to write. There are hundreds of stylesheets in this book, including examples for every XSLT element, function, and operator defined by XSLT and XPath. The first chapters of the book are prose that explain how stylesheets work and what you need to learn to be productive with XSLT. Once you’re comfortable with that material, you can use the rest of the book as a dictionary-style reference.

I Don’t Care Which Standards-Compliant Tools You Use

My job as an author and a teacher is to show you how to use standards-compliant tools to simplify your life. I’m not here to sell you a parser, an XSLT processor, a toaster, or anything else, so please use whatever tools you like. I encourage you to take a look at all of the tools out there and find your own preferences. As I wrote this edition of the book, I used four processors to test the examples:

• Almost all of the examples were tested with Michael Kay’s excellent Saxon XSLT processor. The open source edition of Saxon supports all of the XSLT 2.0, XPath 2.0 and XQuery 1.0 specs except for the schema-specific functions. As the editor of the XSLT 2.0 specification, Dr. Kay’s processor is currently the most complete implementation of XSLT 2.0.

  Saxon-B (the basic processor without schema support) is available here: http://saxon.sourceforge.net/. The SourceForge project page is at http://sourceforge.net/projects/saxon. Saxon is available in Java and .NET versions.

  There is also a commercial version of Saxon that includes full schema support. For more information on Saxon-SA, which is the schema-aware version, visit http://www.saxonica.com/.

• The XSLT engine from Altova XML Spy was also used for all of the XSLT 2.0 examples. The Altova XSLT engine, although not open source, does provide complete schema support in a no-cost product. The license for the Altova engine currently allows you to redistribute it with your own code. To get the engine and the license terms, visit http://www.altova.com/altovaxml.html.

• Apache’s Xalan XSLT engine supports almost all of the XSLT 1.0 examples in the book. (The XSLT 1.0 stylesheets that it doesn’t support are ones that use extensions written for other processors.) It’s also a forwards-compatible XSLT processor, so it can work with XSLT 2.0 stylesheets.


• Microsoft’s .NET framework supports XSLT 1.0, as does the MSXSL utility. One significant addition to this edition is a focus on the Microsoft platform. In
addition to testing all of the XSLT 1.0 samples with the Microsoft tools, there are also XSLT extensions written in C# and EcmaScript.

The MSXSL XSLT processor is available from the Microsoft XML downloads page, http://msdn.microsoft.com/XML/XMLDownloads/default.aspx. There is also an XSLT processor embedded in the .NET framework; it’s part of the System.Xml.Xsl namespace.

**XSLT Is a Tool, Not a Religion**

An old adage says that to a person with a hammer, everything looks like a nail. I don’t claim that XSLT is the solution to every business problem you’ll encounter. Chapter 1 discusses reasons why XML and XSLT were created and the design decisions behind XSLT, and it tries to identify the kinds of problems XSLT is designed to solve. All chapters in this book illustrate common scenarios in which XSLT is extremely powerful and useful.

That being said, if a particular tool does something better than XSLT does, then by all means, use that other tool. For example, XSLT has functions for sorting and grouping. If the data you’re transforming comes from a relational database, it’s probably far more efficient to use the **ORDER BY** and **GROUP BY** features of your database instead of sorting and grouping with XSLT. XSLT is a powerful addition to your tool box, but that doesn’t mean you should throw out all your other tools.

**You Shouldn’t Migrate All of Your Stylesheets Just Because There’s a New Version of XSLT**

Anytime a new version of a language, standard, or software package comes along, deciding when or if to migrate to the new features depends on your application. If you’ve built a web application in which you use a web browser to process XSLT stylesheets on the client side, you can’t migrate to XSLT 2.0 until all the major browsers support XSLT 2.0. That’s going to be a while. On the other hand, if you use XSLT to transform your data and then send the transformed data to the client, you can use XSLT 2.0 right away. With very few exceptions, anything that worked in XSLT 1.0 works in XSLT 2.0. We cover migration in Appendix G.

XSLT 2.0 and XPath 2.0 have many new features that make your stylesheets easier to write, easier to maintain, and much more powerful. It’s definitely worth your time to investigate the new features to see how many of them you can use.

**How This Book Is Organized**

XSLT 2.0 has added significant new features to the language, many of which are related to the changes in XPath 2.0. The biggest challenge I had as an author was figuring out how to organize the book. One approach would have been to make this an XSLT 2.0
book, writing under the assumption that everyone would migrate to XSLT 2.0 as soon as possible. I don’t believe that will happen, so I didn’t go that way. Instead, I tried to cover everything in terms of common tasks, things you’ll probably have to do with XSLT. If there are new features in XSLT 2.0 that apply to those tasks, I mention them after explaining the concepts behind the stylesheets. Usually XSLT 2.0 makes your life much easier, so I begin the discussion by pointing out that if you’re using XSLT 2.0, you’ve got a simpler option.

As with the first edition, this book has two parts: a series of prose chapters that cover concepts and tasks, followed by a series of appendixes that form a reference to all of the elements, functions, operators, and other details you’ll need as you write stylesheets. Once you’re comfortable with XSLT, you can use the appendixes as a dictionary of all things related to XSLT and XPath.

The book contains the following chapters:

Chapter 1, Getting Started
Covers the basics of XML and discusses how to install the stylesheet engines used in this book.

Chapter 2, The Obligatory Hello World Example
Takes a look at an XML-tagged “Hello World” document, then examines stylesheets that transform it into other things.

Chapter 3, XPath: A Syntax for Describing Needles and Haystacks
Covers the basics of XPath, the language used to describe parts of an XML document. This chapter includes an in-depth discussion of the many changes introduced in XPath 2.0.

Chapter 4, Creating Output
Discusses the basics of creating output, including extracting text, copying information, and numbering things.

Chapter 5, Branching and Control Elements
Discusses the logic elements of XSLT (<xsl:if> and <xsl:choose>) and how they work. Also covers the new if operator in XPath 2.0.

Chapter 6, Creating Links and Cross-References
Covers the different ways to build links between elements in XML documents. Using XPath to describe relationships between related elements is also covered.

Chapter 7, Sorting and Grouping Elements
Goes over the <xsl:sort> element and discusses various ways to sort elements in an XML document. It also talks about how to do grouping with various XSLT elements and functions. Grouping is much simpler in XSLT 2.0; the new grouping features are covered in this chapter as well.
Chapter 8, *Combining Documents*

Discusses the `document()` function, which allows you to combine several XML documents, then write a stylesheet that works against the collection of documents. Related functions from XSLT 2.0 are also featured.

Chapter 9, *Extending XSLT*

Explains how to write extension elements and extension functions. Although XSLT and XPath are extremely powerful and flexible, there are still times when you need to do something that isn’t provided by the language itself.

The last section of the book contains reference information:

Appendix A

An alphabetical listing of all the elements defined by XSLT, with examples for those elements and how they were designed to be used.

Appendix B

A listing of various aspects of XPath, including datatypes, axes, node types, and operators.

Appendix C

An alphabetical listing of all the functions defined by XPath and XSLT.

Appendix D

Provides a brief overview of XML Schema. One of the additions to XSLT 2.0 is the ability to use XML Schemas to define datatypes and validate XML structures against them.

Appendix E

Covers the syntax and features of the regular expression language used by XPath 2.0 and XSLT 2.0.

Appendix F

Provides a handy listing of all the formatting codes used in XSLT and XPath.

Appendix G

Lists a number of considerations and approaches for migrating to XSLT 2.0.

Glossary

A glossary of terms used in XSLT, XPath, and XML in general.

**Conventions Used in This Book**

Items appearing in this book are sometimes given a special appearance to set them apart from the regular text. Here’s how they look:

*Italic*

Used for citations of books and articles, commands, email addresses, introduction of terms, and URLs
**Constant width**
Used for literals, constant values, code listings, and XML markup

**Constant-width bold**
Used to indicate user input

**Constant-width italic**
Used for replaceable parameter and variable names

This icon represents a tip, suggestion, or general note.

This icon represents a warning or caution.

[1.0]
This text represents information that applies only to XSLT 1.0 and XPath 1.0.

[2.0]
This text represents information that is new in XSLT 2.0 and XPath 2.0.

[2.0 – Schema]
This text represents information that applies to schema-aware XSLT 2.0 processors.

**How to Contact Us**

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

The web site for this book lists examples, errata, and plans for future editions. You can access this page at:
Acknowledgments for the Second Edition

I want to thank Jeni Tennison for being the lead reviewer of this edition. Her ability to see through to the essence of a problem and point out the simplest and most elegant way to solve it is astounding. I have blisters from smacking my forehead as I read her review comments, thinking at the time, “Of course! I should have seen that right away.” Jeni, thank you.

I also benefited from Priscilla Walmsley’s excellent review, especially in the appendixes that cover all the elements and functions in XSLT, XPath, and XQuery. The examples and terminology in those sections are far more useful and correct as a result.

A big thanks to Michael Kay for providing a copy of Saxon-SA to test the schema examples in the book. The entire XSLT community owes him an enormous debt for making the XSLT 2.0 spec robust, readable, and complete, and for writing the Saxon XSLT engine.

This book was written entirely in DocBook, a very powerful XML vocabulary for publishing. Two books have been invaluable as I’ve worked with DocBook. The first is O’Reilly’s *DocBook: The Definitive Guide*, written by Norm Walsh and Leonard Mueller (available online at http://www.oreilly.com/catalog/docbook/chapter/book/docbook.html). If you want to know anything about DocBook, this is the place to look. The open source community also maintains an extremely sophisticated set of XSLT stylesheets that transform DocBook into a variety of other formats. For help in using the DocBook XSL, Bob Stayton’s *DocBook XSL: The Complete Guide* (Sagehill Enterprises; available online at http://sagehill.net/book-description.html) was invaluable. Thanks to all three of these great authors.
I also want to thank the people I’ve worked with over the last few years. The IBM developerWorks team is still a great influence on me. I’ll always think of myself as part of the developerWorks family. During my time with IBM’s Developer Skills organization, I had the great pleasure of working with an incredibly talented team. That group is paid to give away as much knowledge as possible, along with free software to professors and students around the world. Finally, I want to thank the members of my current team in IBM’s Software Group Strategy organization. I’m very happy to be working again for Dirk Nicol, the father of developerWorks.

I will resist the temptation to name names here in fear of forgetting someone. I hope all of you know how much you mean to me, and how much I’ve learned from all of you.

Finally, I want to thank Simon St.Laurent for his guidance on the second edition. Both of us were nervous about figuring out how to add XSLT 2.0 and XPath 2.0 to this book without creating a 5,000 page tome. Unfortunately, I also relied on Simon’s patience as portions of the book took far longer than either of us had hoped. Simon, you’re the best.

Acknowledgments from the First Edition

First and foremost, I’d like to thank the reviewers of this book. David Marston of Lotus was the lead reviewer; David, thank you so much for your comments, wisdom, and knowledge. Along the way, I also got a lot of good feedback and encouragement from Tony Colle, Slavko Malesvic, Dr. Joe Molitoris, Shane O’Donnell, Andy Piper, Sreenivasa Rama Rao, Mike Riley, and Willie Wheeler. This book is significantly better because of your comments and other efforts.

I’d also like to thank my teammates at developerWorks for encouraging me to undertake this project. Taking on an additional full-time job hasn’t been easy, but their advice, flexibility, and understanding as I’ve tried to balance my responsibilities has been invaluable. Even more valuable is the fact that I’m surrounded by some of the most interesting, creative, and remarkable people I’ve ever known. You guys rule.

For the times I’ve been at home (in Raleigh, North Carolina), I’ve depended on my nutritional advisors at Schiano’s Pizza: “Hey, you want your usual?” (Slight pause.) “Yeah, that’d be great, thanks.” Nothing’s as comforting as a couple of slices. If you’re within a day’s drive of Raleigh, I strongly encourage you to visit.

Finally, I’d like to thank the staff at O’Reilly, especially Laurie Petrycki and Simon St.Laurent. Laurie, thank you for convincing me to take on this project and for sticking with me when my ability to find the time to write was in doubt. Simon, I’ve enjoyed reading your books for years; it’s been an honor to work with you. Your guidance, technical insight, patience, and suggestions were invaluable.

Thanks so much to all of you!
In this chapter, we review the design rationale behind XSLT and XPath and discuss the basics of XML. We also talk about other web standards and how they relate to XSLT and XPath. We conclude the chapter with a brief discussion of how to set up an XSLT processor on your machine so you can work with the examples throughout the book.

The Design of XSLT

XML went from working group to entrenched buzzword in record time. Its flexibility as a language for presenting structured data made it the lingua franca for data interchange. Early adopters used programming interfaces such as the Document Object Model (DOM) and the Simple API for XML (SAX) to parse and process XML documents. As XML became mainstream, however, it was clear that the average web citizen couldn’t be expected to hack Java, Visual Basic, Perl, or Python code to work with documents. What was needed was a flexible, powerful, yet relatively simple language capable of processing XML.

What the world needed was XSLT.

XSLT, the Extensible Stylesheet Language for Transformations, is an official recommendation of the World Wide Web Consortium (W3C). It provides a flexible, powerful language for transforming XML documents into something else, such as an HTML document, another XML document, a Portable Document Format (PDF) file, a Scalable Vector Graphics (SVG) file, a Virtual Reality Modeling Language (VRML) file, Java code, a flat text file, a JPEG file, or most anything you want. You write an XSLT stylesheet to define the rules for transforming an XML document, and the XSLT processor does the work.

The W3C has defined two families of standards for stylesheets. The oldest and simplest is Cascading Style Sheets (CSS), a mechanism used to define various properties of markup elements. Although CSS can be used with XML, it is most often used to style HTML documents. I can use CSS properties to define certain elements to be rendered in blue, or in 58-point type, or in boldface. That’s all well and good, but there are many things that CSS can’t do.
• CSS can’t change the order in which elements appear in a document. If you want to sort certain elements or filter elements based on a certain property, CSS won’t do the job.

• CSS can’t do computations. If you want to calculate and output a value (maybe you want to add up the numeric value of all `<price>` elements in a document), CSS won’t do the job.

• CSS can’t combine multiple documents. If you want to combine 53 purchase order documents and print a summary of all items ordered in those purchase orders, CSS won’t do the job.

Don’t take this section as a criticism of CSS; XSLT and CSS were designed for different purposes. One fairly common use of XSLT is to generate an HTML document that uses CSS. See “The XPath View of an XML Document” in Chapter 3 for an example that uses XSLT to generate CSS classes, and then uses those classes to format the HTML elements.

XSLT was created to be a more powerful, flexible language for transforming documents. In this book, we go through all the features of XSLT and discuss each of them in terms of practical examples. Some of XSLT’s design goals specify that:

• An XSLT stylesheet should be an XML document. This means that you can write a stylesheet that transforms a second stylesheet into another stylesheet. This kind of recursive thinking is common in XSLT.

• The XSLT language should be based on pattern matching. Most of our stylesheets consist of rules (called templates in XSLT) used to transform a document. Each rule says, “When you see part of a document that looks like this, here’s how you convert it into something else.” This is probably different from any programming you’ve previously done.

• XSLT should be designed to be free of side effects. In other words, XSLT is designed to be optimized so that many different stylesheet rules could be applied simultaneously. The biggest impact of this is that variables can’t be modified. Once a variable is bound, you can’t change its value; if variables could be changed, then processing one stylesheet rule might have side effects that impact other stylesheet rules. This is almost certainly different from any programming you’ve previously done.

XSLT is heavily influenced by the design of functional programming languages, such as Lisp, Scheme, and Haskell. These languages also feature immutable variables. Instead of defining the templates of XSLT, functional programming languages define programs as a series of functions, each of which generates a well-defined output (free from side effects, of course) in response to a well-defined input. The goal is to execute the instructions of a given XSLT template without affecting the execution of any other XSLT template.
• Instead of looping, XSLT uses iteration and recursion. Given that variables can’t be changed, how do you do something like a for or do-while loop? XSLT uses two equivalent techniques: iteration and recursion. Iteration means that you can write an XSLT template that says, “Get all the things that look like this, and here’s what I want you to do with each of them.” Although that’s different from a do-while loop, usually what you do in a procedural language is something like, “Do this while there are any items left to process.” In that case, iteration does exactly what you want.

Recursion takes some getting used to. If you must implement something like a for statement (for i=1 to 10 do, for example), recursion is the way to go. There are a number of examples of recursion throughout the book; you can flip ahead to “Using Recursion to Do Most Anything” in Chapter 5 for more information.

Given these design goals, what are XSLT’s strengths? Here are some scenarios:

• Your web site needs to deliver information to a variety of devices. You need to support ordinary desktop browsers, as well as pagers, mobile phones, and other low-resolution, low-function devices. It would be great if you could create your information in structured documents, then transform those documents into all the formats you need.

• You need to exchange data with your partners, but all of you use different database systems. It would be great if you could define a common XML data format, then transform documents written in that format into the import files you need (SQL statements, comma-separated values, etc.).

• To stay on the cutting edge, your web site gets a complete visual redesign every few months. Even though things such as server-side includes and CSS can help, they can’t do everything. It would be great if your data were in a flexible format that could be transformed into any look and feel, simplifying the redesign process.

• You have documents in several different formats. All the documents are machine-readable, but it’s a hassle to write programs to parse and process all of them. It would be great if you could combine all of the documents into a single format, then generate summary documents and reports based on that collection of documents. It would be even better if the report could contain calculated values, automatically generated graphics, and formatting for high-quality printing.

Throughout the book, we’ll demonstrate XSLT solutions for problems just like these. Most chapters focus on particular techniques, such as sorting, grouping, and generating links between pieces of data, although we’ll start with a gentle introduction to the basics.

[2.0] The Design of XSLT 2.0

XSLT 2.0 is a major enhancement to the language. XSLT 2.0 uses XPath 2.0, which itself went through many significant changes. The gap between XSLT 1.0/XPath 1.0
and XSLT 2.0/XPath 2.0 was a little over seven years (November 16, 1999 to January 23, 2007). There were two major requirements that led to the monumental amount of work required to create XSLT 2.0 and XPath 2.0:

**Support for XML Schema**
XSLT and XPath now support XML Schema, which means nodes and variables can have datatypes. We can define a value to be of type `xs:dateTime`, and the XSLT processor will enforce that requirement. All XSLT 2.0 processors support the basic XML Schema datatypes. A **schema-aware processor** also supports custom datatypes. If we have a datatype named `purchaseOrder`, we can use a schema-aware processor to work with values of that type.

**Integration with XQuery**
The initial work for XQuery began in 1998, and version 1.0 became a W3C Recommendation on January 23, 2007. XQuery 1.0 and XPath 2.0 share a common data model, functions, and operators. Coordinating the efforts of the XQuery, XPath, and XSLT working groups must have been a challenge.

The birthing pains of XSLT 2.0 and XPath 2.0 are behind us now, and we have a more powerful language for transforming documents. We’ll discuss the changes to the language as they’re relevant to our discussion of common tasks that you’ll probably want to do with XSLT. All of the technical details are covered in the appendixes.

**XML Basics**
Almost everything we do in this book deals with XML documents. XSLT stylesheets are XML documents themselves, and they’re designed to transform an XML document into something else. If you don’t have much experience with XML, we’ll review the basics here. For more information on XML, check out Erik T. Ray’s *Learning XML* (O’Reilly, 2001) and Elliotte Rusty Harold and W. Scott Means’s *XML in a Nutshell* (O’Reilly, 2001).

**XML’s Heritage**
XML’s heritage is in the Standard Generalized Markup Language (SGML). Created by Dr. Charles Goldfarb in the 1970s, SGML is widely used in high-end publishing systems. Unfortunately, SGML’s perceived complexity prevented its widespread adoption across the industry (SGML also stands for “sounds great, maybe later”). SGML got a boost when Tim Berners-Lee based HTML on SGML. Overnight, the whole computing industry was using a markup language to build documents and applications.

The problem with HTML is that its tags were designed for the interaction between humans and machines. When the Web was invented in the late 1980s, that was just fine. As the Web moved into all aspects of our lives, HTML was asked to do lots of strange things. We’ve seen HTML tags for awkward table structures, 1-pixel
GIFs, and other nonsense just to get the page to look right in the browser. XML is designed to get us out of this rut and back into the world of structured documents.

Whatever its limitations, HTML is the most popular markup language ever created. Given its popularity, why do we need XML? Consider this extremely informative HTML element:

```html
<td>12304</td>
```

What does this fascinating piece of content represent?

- Is it the postal code for Schenectady, New York?
- Is it the number of light bulbs replaced each month in Las Vegas?
- Is it the number of Volkswagens sold in Hong Kong last year?
- Is it the number of tons of steel in the Sydney Harbour Bridge?

The answer: maybe, maybe not. The point of this silly example is that there’s no structure to this data. Even if we include the entire table, it takes intelligence (real, live intelligence, the kind between your ears) to make sense of this. If you saw this cell in a table next to another cell that contained the text “Schenectady,” and the heading above the table read “Postal Codes for the State of New York,” then as a human being, you could interpret the contents of this cell correctly. On the other hand, if you wanted to write a piece of code that took any HTML table and attempted to determine whether any of the cells in the table contained postal codes, you’d find that difficult, to say the least.

Most HTML pages have one goal in mind: the appearance of the document. Veterans of the markup industry know that this is definitely not the way to create content. The separation of content and presentation is a long-established tenet of the publishing industry; unfortunately, most HTML pages aren’t even close to approaching this ideal.

An XML document should contain information, marked up with tags that describe what all the pieces of information are, as well as the relationship between those items. Presenting the document (also known as rendering) involves rules and decisions separate from the document itself. As we work through dozens of sample documents and applications, you’ll see how delaying the rendering decisions as long as possible has significant advantages.

Let’s look at another marked-up document. Consider this:

```xml
<?xml version="1.0"?>
<postalcodes>
  <title>Most-used postal codes in November 2000</title>
  <item>
    <city>Schenectady</city>
    <postalcode>12304</postalcode>
    <usage-count>2039</usage-count>
  </item>
  <item>
    <city>Kuala Lumpur</city>
    <postalcode>57000</postalcode>
  </item>
</postalcodes>
```
Although we’re still in the realm of contrived examples, it would be fairly easy to write a piece of code to find the postal codes in any document that used this set of tags (as opposed to HTML’s `<table>`, `<tr>`, `<td>`, etc.). Our code would look for the contents of any `<postalcode>` elements in the document. (Not to get ahead of ourselves here, but writing an XSLT stylesheet to do this might take all of 30 minutes, including a 25-minute nap.) A well-designed XML document identifies each piece of data in the document and models the relationships between those pieces of data. This means we can be confident that we’re processing an XML document correctly.

Again, the key idea here is that we’re separating content from presentation. Our XML document clearly delineates the pieces of data and puts them into a format we can parse easily. In this book, we illustrate a number of techniques for transforming this XML document into a variety of formats. Among other things, we can transform the item `<postalcode>12304</postalcode>` into `<td>12304</td>`.

**XML Document Rules**

Continuing our trip through the basics of XML, there are several rules you need to keep in mind when creating XML documents. All stylesheets we develop in this book are themselves XML documents, so all the rules of XML documents apply to everything we do. The rules are pretty simple, even though the vast majority of HTML documents don’t follow them.

One important point: the XML 1.0 specification makes it clear that when an XML parser finds an XML document that breaks the rules, the parser is supposed to throw an exception and stop. The parser is not allowed to guess what the document structure should actually be. This specification avoids recreating the HTML world, where lots of ugly documents are still rendered by the average browser.

**An XML document must be contained in a single element**

The first element in your XML document must contain the entire document. That first element is called the *document element* or the *root element*. If more than one document element is in the document, the XML parser throws an exception. This XML document is perfectly legal:

```xml
<?xml version="1.0"?>
<greeting>
```
Hello, World!
</greeting>

To be precise, this document is well-formed. XML documents are described as well-formed and valid (we’ll define those terms in a minute). This XML document isn’t legal at all:

```xml
<?xml version="1.0"?>
<greeting>
  Hello, World!
</greeting>
<greeting>
  Hey, Y’all!
</greeting>
```

There are two root elements in this document, so an XML parser refuses to process it. Also, be aware that the XML declaration (the `<<?xml version="1.0"?> part; more on this later) isn’t an element at all.

**All elements must be nested**

If you start one element inside another, you have to end it there, too. An HTML browser is happy to render this document:

```html
<b>I really, <i>really</i></b> like XML.</i>
```

But an XML parser will throw an exception when it sees this document. If you want the same effect, you would need to code this:

```xml
<b>I really, <i>really</i></b></i> like XML.</i>
```

**All attributes must be quoted**

You can quote the attributes with either single or double quotes. These two XML tags are equivalent:

```xml
<a href="http://www.oreilly.com">
<a href='http://www.oreilly.com'>
```

If you need to define an attribute that contains single or double quotes, you can use one style of quote inside the other. If you need both single and double quotes in an attribute, use the predefined entities &quot; for double quotes and use &apos; for single quotes:

```xml
<book title="XSLT, Second Edition" publisher="O'Reilly/>
<book title="XSLT, Second Edition" publisher='O''reilly'/>
```

One more note: XML doesn’t allow attributes without values. In other words, HTML elements such as `<ol compact>` aren’t valid in XML. To code this element in XML, you’d have to give the attribute a value, as in `<ol compact="compact">`. (You have to do things this way in XHTML as well.)
XML tags are case-sensitive

In HTML, `<h1>` and `<H1>` are the same. In XML, they’re not. If you try to end an `<h1>` element with `</H1>`, the parser will throw an exception.

All end tags are required

This is another area where most HTML documents break. Your browser doesn’t care whether you don’t have a `</p>` or `</br>` tag, but your XML parser does.

Empty tags can contain the end marker

In other words, these two XML fragments are identical:

```
<lily age="13"></lily>
```

```
<lily age="13"/>
```

Notice that there is nothing, not even whitespace, between the start tag and the end tag in the first example; that’s what makes this an empty tag.

XML declarations

Some XML documents begin with an XML declaration, which is a line similar to this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

If no encoding is specified, the XML parser assumes you’re using UTF-8 or UTF-16. UTF, the Unicode Transformation Format, is a Unicode standard that uses different numbers of bytes to represent virtually every character and ideograph from the world’s languages. Be aware that each parser supports a different set of encodings, so you need to check your parser’s documentation to find out what your options are.

Document Type Definitions (DTDs) and XML Schemas

All of the rules we’ve discussed so far apply to all XML documents. In addition, you can use DTDs and Schemas to define other constraints for your XML documents. DTDs and Schemas are metalanguages that let you define the characteristics of an XML vocabulary. For example, you might want to specify that any XML document describing a purchase order must begin with a `<po>` element, and the `<po>` element in turn contains a `<customer-id>` element, one or more `<item-ordered>` elements, and an `<order-date>` element. In addition, each `<item-ordered>` element must contain a `part-number` attribute and a `quantity` attribute.

Here’s a sample DTD that defines the constraints we just mentioned:

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

<!ELEMENT po (customer-id , item-ordered+ , order-date)>
<!ELEMENT customer-id (#PCDATA)>
```

And here’s an XML Schema that defines the same document type:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="po">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="customer-id"/>
        <xsd:element ref="item-ordered" maxOccurs="unbounded"/>
        <xsd:element ref="order-date"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="customer-id" type="xsd:string"/>

  <xsd:element name="item-ordered">
    <xsd:complexType>
      <xsd:attribute name="part-number" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="quantity" use="required" type="xsd:integer"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="order-date">
    <xsd:complexType>
      <xsd:attribute name="day" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxInclusive value="31"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="month" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxInclusive value="12"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
Schemas have two significant advantages over DTDs:

*They can define datatypes and other complex structures that are difficult or impossible to do in a DTD*

In the previous example, we defined various constraints for the data in our XML documents. We defined that the *day* attribute must be an integer between 1 and 31, and the *month* attribute must be an integer between 1 and 12. We also used a regular expression to define a *part-number* attribute as a five-digit number, optionally followed by a dash and a four-digit number. None of those things are possible in a DTD. Schemas are far more powerful than DTDs; see Appendix D for an overview of schemas and what they can do.

*Schemas are themselves XML documents*

Since they are XML documents, we can write XSLT stylesheets to manipulate them. For example, it would be useful to create a graphical representation of an XML Schema. We could create a hierarchical diagram to indicate which elements could appear inside other element. XML Schema also provides the `<xsd:annotation>` and `<xsd:documentation>` elements. Those elements let us add as much documentation as we want inside the schema itself. We could then use a stylesheet to transform the schema into an HTML document or PDF file, using the relationships between elements, attributes, datatypes, and other information to generate highly structured information.

The best way to define the `<order-date>` attribute would be to use the XML Schema `xsd:date` datatype:

```xml
<xsd:element name="order-date" type="xsd:date"/>
```

In the DTD, we separated the date into three parts so it could be sorted or formatted in different ways. With the `xsd:date` datatype, the schema ensures that the date is valid; we can use a variety of functions to sort or format the date in different ways. (We'll discuss those functions in “[2.0] Formatting Dates and Times” in Chapter 4.)

**Well-formed versus valid documents**

Any XML document that follows the rules described here is said to be well-formed. In addition, if an XML document references a set of rules that define how the document
is structured (either a DTD or an XML Schema), and it follows all those rules, it is said to be a valid document.

All valid documents are well-formed; on the other hand, not all well-formed documents are valid.

Be aware that XML Schema validation can be done partially; XML Schema allows us to define parts of the document that should not be validated at all. On the other hand, DTD validation fails if any part of an XML document doesn’t match the DTD.

Tags versus elements

Although many people use the two terms interchangeably, a tag is different from an element. A tag is the text between (and including) the angle brackets (< and >). There are start tags, end tags, and empty tags. A tag consists of an element name and, if it is a start tag or an empty tag, some optional attributes. (Unlike other markup languages, end tags in XML cannot contain attributes.) An element consists of its start and end tags and everything in between. This might include text, other elements, and comments, as well as other things such as entity references and processing instructions.

Namespaces

A final XML topic we’ll mention here is namespaces. Namespaces are designed to distinguish between two tags that have the same name. For example, if we have an online bookstore, we could design an XML vocabulary for books. When we ship an order to a customer, the postal service requires the customer’s address to be in a certain format. It’s likely that both vocabularies will define a <title> element. Our <title> element refers to the title of a book, while the shipping company’s <title> element refers to the courtesy title of a customer (Mr., Ms., Mrs., etc.). An XML order document refers to both books and customers, so we’ll use a namespace to distinguish between the two <title> elements. Namespaces are declared as follows:

```xml
<xyz xmlns:books="http://www.myco.com/books"
     xmlns:addr="http://www.usps.com/addresses">
```

In this example, the xmlns:books attribute associates the prefix books with one namespace, and the xmlns:addr attribute associates the paintings prefix with another namespace. This means that a title element from the books namespace would be coded as <books:title>, while a title element from the addr namespace would be referred to as <addr:title>.

I mention namespaces here primarily because all XSLT elements we use in this book are prefixed with the xsl namespace prefix. All stylesheets we write begin like this:

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```
(Obviously a stylesheet that uses the features of XSLT 2.0 starts with version="2.0".) This opening associates the xsl namespace prefix with the string http://www.w3.org/1999/XSL/Transform. The value of the namespace prefix doesn’t matter; we could start our stylesheets like this:

```xml
<?xml version="1.0"?>
<pdq:stylesheet version="1.0"
xmlns:pdq="http://www.w3.org/1999/XSL/Transform">
```

What matters is the string to which the namespace prefix is mapped. Also keep in mind that all XSLT stylesheets use namespace prefixes to process the XML elements they contain. By default, anything that doesn’t use the xsl namespace prefix is not processed—instead, it’s written to the result tree. We’ll discuss these topics in more detail as we go through the book.

[2.0] Datatypes

XSLT 2.0 provides support for most of the datatypes defined in XML Schema. XSLT 2.0 also defines new datatypes for durations. For example, we can define an XSLT variable and specify that its datatype is xs:integer or xs:dateTime. If we’re using a schema-aware XSLT 2.0 processor, we can define our own datatypes and use those just like all the datatypes defined by XML Schema and XSLT 2.0. We cover datatypes and schemas in Chapter 3.

Programming Interfaces for XML: DOM, SAX, and Others

The two most popular APIs used to parse XML documents are the Document Object Model (DOM) and the Simple API for XML (SAX). DOM is an official recommendation of the W3C (available at http://www.w3.org/TR/REC-DOM-Level-1), while SAX is a de facto standard created by David Megginson and others on the XML-DEV mailing list (http://lists.xml.org/archives). We’ll discuss these two APIs briefly here. We won’t use them much in this book, but learning more about them will give you some insight into how most XSLT processors work.

See http://www.saxproject.org/ for the SAX standard. If you’d like to learn more about the XML-DEV mailing list, send email to mailto:xml-dev-subscribe@lists.xml.org. You can also check out http://lists.xml.org/archives/xml-dev/ to see the XML-DEV mailing list archives.

DOM

DOM is designed to build a tree view of your document. Remember that all XML documents must be contained in a single element. That single element then becomes the root of the tree. The DOM specification defines several language-neutral interfaces, described here:
Node
This interface is the base datatype of the DOM. Document, Element, Attr, Text, Comment, and ProcessingInstruction all extend the Node interface.

Document
This object contains the DOM representation of the XML document. Given a Document object, you can get the root of the tree (the Document element); from the root, you can move through the tree to find all elements, attributes, text, comments, processing instructions, etc. in the XML document.

Element
This interface represents an element in an XML document.

Attr
This interface represents an attribute of an element in an XML document.

Text
This interface represents a piece of text from the XML document. Any text in your XML document becomes a Text node. This means that the text of a DOM object is a child of the object, not a property of it. The text of an Element is represented as a Text child of an Element object; the text of an Attr is also represented that way.

Comment
This interface represents a comment in the XML document. A comment begins with <!-- and ends with -->. The only restriction on its contents is that two consecutive hyphens (--) can appear only at the start or end of the comment. Other than that, a comment can include anything, such as angle brackets (< >), ampersands (&), and single or double quotation marks (' "').

ProcessingInstruction
This interface represents a processing instruction in the XML document. Processing instructions look like this:

```xml
<?xml-stylesheet href="case-study.xsl" type="text/xsl"?>
```

Processing instructions contain processor-specific information. The PI here (PI is XML jargon—feel free to drop this into casual conversations to impress your friends) is the standard way to associate an XSLT stylesheet with an XML document (more on this in a minute).

When you parse an XML document with a DOM parser, it:

- Creates objects (Elements, Attr, Text, Comments) representing the contents of the document. These objects implement the interfaces defined in the DOM specification.
- Arranges these objects in a tree. Each Element in the XML document has some properties (such as the element’s name) and may also have some children.
- Parses the entire document before control returns to your code. This means that for large documents, there is a long delay before the document is parsed.
The most significant thing about the DOM is that it is based on a tree view of your document. An XSLT processor uses a very similar tree view (with some slight differences, such as the fact that not everything we deal with in XPath and XSLT has the same root element). Understanding how a DOM parser works makes it easier to understand how an XSLT processor views your document.

**A sample DOM tree.** DOM, XSLT, and XPath all use tree structures to represent data from an XML document. For this reason, it’s important to have at least a casual knowledge of how DOM builds a tree structure. Our earlier `<postalcodes>` document is shown as a DOM tree in Figure 1-1.

If we want to perform tasks such as find different parts of our XML document, sort the subtrees based on the first character of the text of the `<postalcode>` element, or select only the subtrees in which the text of the `<usage-count>` element has a numeric value greater than 500, we have to start at the top of the DOM tree and work our way down through the root element’s descendants. When we write XSLT stylesheets, we also start at the root of the tree and work our way down.
To be honest, the DOM tree built for our document is more complicated than our beautiful picture indicates. The whitespace characters in our document (carriage return/line feed, tabs, spaces, etc.) become Text nodes. Normally it’s a good idea to remove this whitespace so the DOM tree won’t be littered with these useless nodes, but I include them here to give you a sense of the XML document’s structure.

**SAX**

The Simple API for XML was developed by David Megginson and others on the XML-DEV mailing list. It has several important differences from DOM:

- The SAX API is interactive. As a SAX parser processes your document, it sends events to your code. You don’t have to wait for the parser to finish the entire document as you do with the DOM; you get events from the parser immediately. These events let you know when the parser finds the start of the document, the start of an element, some text, the end of an element, a processing instruction, the end of the document, etc.

- SAX is designed to avoid the large memory footprint of DOM. In the SAX world, you’re told when the parser finds things in the XML document; it’s up to you to save those things. If you don’t do anything to store the data found by the parser, it goes into the bit bucket.

- SAX doesn’t provide the hierarchical view of the document that DOM does. If you need to know a lot about the structure of an XML document and the context of a given element, SAX isn’t much help. Each SAX event is stateless; that is, a SAX event won’t tell you, “Here’s some text for the `<postalcode>` element I mentioned earlier.” A SAX parser only tells you, “Here’s some text.” If you need to know about an XML document’s structure, you have to keep track of that information yourself.

The best thing about SAX is that it is interactive. Most of the transformations currently done with XSLT take place on the server. As of this writing, most XSLT processors are based on DOM parsers. In the near future, however, we’ll see XSLT processors based on SAX parsers. This means that the processor can start generating results almost as soon as the parse of the source document begins, resulting in better throughput and creating the perception of faster service. Because DOM, XPath, and XSLT all use trees to represent XML documents, DOM is more relevant to our discussions here. Nevertheless, it’s useful to know how SAX parsers work, especially as SAX-based XSLT processors begin to rear their speedy little heads.

**Other programming interfaces**

There are a number of other XML programming interfaces, including JDOM, DOM4J, and StAX. These have two important characteristics:
In-memory versus event-driven

In-memory interfaces, such as DOM, create data structures that represent the XML document. Event-driven interfaces, such as SAX, receive data from the parser as it parses the document.

Push versus pull

A push interface pushes data from the parser to the application. When the parser has some data, it uses a callback interface to push that data to the application. SAX is an example of a push interface. On the other hand, a pull interface is still event-driven, but the application tells the parser when it wants the next event. StAX, the Streaming API for XML, is an example of a pull interface. (StAX is also known as JSR 173.)

There are two other approaches we’ll mention briefly. In data binding, an XML document is transformed into an object. The contents of the original XML document are represented as the properties of that object. Finally, a new parsing technique called non-extractive XML processing creates Virtual Token Descriptors that contain the offset, length, and other information of XML tokens inside the XML file itself.

The Wikipedia entry http://en.wikipedia.org/wiki/XML#Processing/XML_files has more detail on these approaches as well as links to various tools that implement them.

XSLT Standards

XSLT 1.0 is defined in two documents: the XSLT and XPath specifications. XSLT 2.0 and XPath 2.0, on the other hand, are defined in a set of eight documents. We’ll discuss all of those specifications briefly in the next section.

XSL transformations (XSLT) version 1.0

The original standard became a recommendation of the W3C on November 16, 1999. The spec lives here: http://www.w3.org/TR/xslt.

XML path language (XPath) version 1.0

XPath 1.0 became a standard on the same day as XSLT 1.0. XPath began as part of XSLT. If we’re going to write a stylesheet to transform an XML document, we have to have a syntax for describing different parts of that document. As the development of XSLT continued, it became obvious that XPath was useful for a variety of applications, so XPath became a separate standard. You can find the definition of XPath 1.0 at http://www.w3.org/TR/xpath.

XSL transformations (XSLT) version 2.0

The basic definition of XSLT 2.0 is at http://www.w3.org/TR/xslt20/. This document defines the elements of XSLT 2.0 and a variety of functions and also defines how XSLT 2.0 processes an XML document.
**XML path language (XPath) version 2.0**

The basic definition of XPath 2.0 is at [http://www.w3.org/TR/xpath20/](http://www.w3.org/TR/xpath20/). XPath 2.0 is built on top of several other documents; we’ll list those next.

**XQuery 1.0 and XPath 2.0 Data Model (XDM)**

This spec defines the way XPath 2.0, XSLT 2.0, and XQuery 1.0 organize data. It defines the information contained in the input to an XSLT 2.0 or XQuery 1.0 processor. It also defines all of the legal values for expressions in XPath 2.0, XSLT 2.0, and XQuery 1.0. You can find the spec at [http://www.w3.org/TR/xpath-datamodel/](http://www.w3.org/TR/xpath-datamodel/).

**XQuery 1.0 and XPath 2.0 functions and operators**

This spec, also known as F&O, defines all of the functions and data operators available in XPath 2.0 and XQuery 1.0. For example, the spec defines how an `xs:yearMonthDuration` can be divided by an `xs:double` value. It also defines the `matches()` function, which determines if a value matches a regular expression. The spec is available at [http://www.w3.org/TR/xpath-functions/](http://www.w3.org/TR/xpath-functions/).

**XQuery 1.0 and XPath 2.0 formal semantics**

The formal semantics spec defines a precise meaning to all of the legal expressions in XPath 2.0 and XQuery 1.0. The XQuery 1.0 and XPath 2.0 Data Model is used in those precise definitions. Possibly the least useful spec to XSLT programmers, it’s available at [http://www.w3.org/TR/xquery-semantics/](http://www.w3.org/TR/xquery-semantics/).

**XSLT 2.0 and XQuery 1.0 serialization**

The serialization spec defines how to take an instance of the XQuery 1.0/XPath 2.0 Data Model and serialize it. For the examples in this book, we’ll usually take the results generated by our XSLT stylesheet and write them to a file; the serialization spec defines how that process works. The spec is available at [http://www.w3.org/TR/xslt-xquery-serialization/](http://www.w3.org/TR/xslt-xquery-serialization/).

**XQuery 1.0: an XML query language**

XQuery 1.0 is a separate language that is based on XPath and other query languages. It is a superset of XPath 2.0. We won’t cover XQuery in any detail in this book, but be aware that the data model, the functions, and the operators of XPath 2.0 are shared by XQuery. See [http://www.w3.org/TR/xquery/](http://www.w3.org/TR/xquery/) for the complete details.

**XML syntax for XQuery 1.0 (XQueryX)**

One of the requirements of the XQuery working group was to provide an XML syntax for the language. XQueryX provides that syntax. It maps the XQuery grammar into XML tags. As such, it is not particularly easy, or convenient for humans, but it can be...
very useful for various tools and utilities. The spec is available at http://www.w3.org/TR/xqueryx.

**XML Standards**

When we talk about writing stylesheets, we’ll work with two standards: XSLT and XPath. XSLT defines a set of primitives used to describe a document transformation, while XPath defines a syntax for describing locations in XML documents. When we write stylesheets, we’ll use XSLT to tell the processor what to do, and we’ll use XPath to tell the processor what document to do it to. Both standards are available at the W3C’s web site; see http://www.w3.org/TR/xslt and http://www.w3.org/TR/xpath for more information.

There are other XML-related standards, of course. We’ll discuss them here briefly, with a short mention of how (or whether) they relate to our work with XSLT and XPath.

**XML 1.0**

The foundation upon which everything else is built. See http://www.w3.org/TR/REC-xml.

**XML 1.1**

You can find the XML 1.1 standard at http://www.w3.org/TR/xml11/.

**The Extensible Stylesheet Language (XSL)**

Also called the Formatting Objects specification or XSL-FO, this standard deals with rendering XML elements. Although most people think of rendering as formatting for a browser or a printed page, researchers use the specification to render XML elements as Braille or as audio files. (That being said, the main market for this technology is in producing high-quality printed output.) As of this writing, the latest version of XSL is 1.1. A couple of the examples in this book use formatting objects and the Apache XML Project’s Formatting Object to PDF translator (FOP) tool; see http://xml.apache.org/fop for more information on FOP. For more information on XSL, see http://www.w3.org/TR/xsl.

**XML Schemas**

In our earlier examples, we had a brief example of an XML Schema. Part 1 of the specification deals with XML document structures; it contains XML elements that define what can appear in an XML document. You use these elements to specify which elements can be nested inside others, how many times each element can appear, the attributes of those elements, and other features. Part 2 of the specification defines basic datatypes used in XML Schemas and rules for deriving new datatypes from existing ones.
The two specifications are available at http://www.w3.org/TR/xmlschema-1 and http://www.w3.org/TR/xmlschema-2. For a good introduction to XML Schemas, see the XML Schema Primer, available at http://www.w3.org/TR/xmlschema-0.

**RelaxNG**

RelaxNG is a simple schema language designed as an alternative to XML Schema. One significant difference between the two is that RelaxNG avoids the many datatype definitions of XML Schema. With RelaxNG, you validate an XML document with datatype definitions imported from elsewhere (including XML Schema, for example). The home page of the OASIS RelaxNG committee is here: http://www.oasis-open.org/committees/relax-ng/. You can find the latest version of the spec as well as a tutorial there.

**Schematron**

Schematron is an elegant way to validate documents. It has a simple syntax (only six elements) and uses XPath to specify patterns in XML documents. The most interesting and most widely used implementation of Schematron is written in XSLT. For more information, including a link to the latest version of the ISO standard for Schematron, visit http://www.schematron.com/.

**The Simple API for XML (SAX)**

The SAX API defines the events and interfaces used to interact with a SAX parser. SAX and DOM are the most common APIs used to work with XML documents. See http://www.saxproject.org/ for the complete specification.

**Document Object Model (DOM)**

The DOM, as we discussed earlier, is a programming API for documents. It defines a set of interfaces and methods used to view an XML document as a tree structure. XSLT and XPath use a similar tree view of XML documents. The home of the DOM is http://www.w3.org/DOM/. This page contains links to all of the W3C Recommendations (Levels 1, 2, and 3) and related documents. The DOM doesn’t affect what we’ll do here, but it’s useful to have a passing knowledge of it. (The XPath data model is similar to the DOM.)

**Namespaces in XML**

As we mentioned earlier, namespaces provide a way to avoid name collisions when two XML elements have the same name. See http://www.w3.org/TR/REC-xml-names/ for the version 1.0 spec; version 1.1 is at http://www.w3.org/TR/REC-xml-names11/.

**Associating stylesheets with XML documents**

It’s possible to reference an XSLT stylesheet within an XML document. This specification uses processing instructions to define one or more stylesheets that should be
used to transform an XML document. You can define different stylesheets to be used for different browsers. See http://www.w3.org/TR/xml-stylesheet for complete information. Here’s the start of an XML document, with two associated stylesheets:

```xml
<?xml version="1.0"?>
<?xml-stylesheet href="docbook/html/docbook.xsl" type="text/xsl"?>
<?xml-stylesheet href="docbook/wap/docbook.xsl"  type="text/xsl" media="wap"?>
```

In this example, the first stylesheet is the default because it doesn’t have a media attribute. The second stylesheet will be used when the User-Agent field from the HTTP header contains the string wap, identifying the requester of a document as a WAP browser. The advantage of this technique is that you can define several different stylesheets within a particular document and have each stylesheet generate useful results for different browser or client types. The disadvantage of this technique is that we’re effectively putting rendering instructions into our XML document, something we prefer to avoid.

**Scalable Vector Graphics (SVG)**

The SVG specification defines an XML vocabulary for vector graphics. Described by some as “PostScript with angle brackets,” it allows you to define images that can be scaled to any size or resolution. See http://www.w3.org/TR/SVG/ for details.

**XML pointer language (XPointer) version 1.0**

XPointer provides a way to identify a fragment of a web resource. It uses XPath to identify fragments. The XPointer Framework is defined at http://www.w3.org/TR/xptr-framework/.

**XML linking language (XLink) version 1.0**

XLink defines an XML vocabulary for linking to other web resources within an XML document. It supports the unidirectional links we’re all familiar with in HTML, as well as more sophisticated links. See http://www.w3.org/TR/xlink/.

**Installing XSLT Processors**

Before we dive in to creating stylesheets, we’ll cover how to install four popular XSLT processors.

**Installing Xalan**

In this section, we’ll go over how to install the Xalan XSLT processor. In the next chapter, we’ll create our first stylesheet and use it to transform an XML document.

The installation process is pretty simple assuming you already have a Java Runtime Environment (JRE) installed on your machine. Although very little of the code we look
at in this book uses Java, the Xalan XSLT processor itself is written in Java. Once you’ve installed the JRE, go to http://xml.apache.org/xalan-j/ and download the latest stable build of the code. (If you’re feeling brave, feel free to download last night’s build instead.)

Once the Xalan .zip or .gzip file is downloaded, unpack it and add three files to your CLASSPATH. The three files include two .jar files for the Xerces parser, and the .jar file for the Xalan stylesheet engine itself. As of this writing, the .jar files are named xalan.jar, xercesImpl.jar, and xml-apis.jar. (There’s a fourth file, bsf.jar, that includes the Bean Scripting Framework, but we’ll use that for extensions only.)

To make sure Xalan is installed correctly, go to a command prompt and type the following command:

```
java org.apache.xalan.xslt.Process
```

This is a Java class, so everything is case-sensitive. You should see an error message like this:

```
java org.apache.xalan.xslt.Process
=xslproc options:
 -IN inputXMLURL
 [-XSL XSLTransformationURL]
 [-OUT outputURL]
 [-LXCIN compiledStylesheetFileNameIn]
 [-LXCOUT compiledStylesheetFileNameOut]
...
```

If you get this message, you’re all set! You’re ready for the next chapter, in which we’ll build our very first XSLT stylesheet.

### Installing Saxon

As of this writing, the most complete open source XSLT 2.0 stylesheet processor is Saxon. Written by Michael Kay, the editor of the XSLT 2.0 spec, it is available at http://saxon.sourceforge.net. When you download the file (currently saxonb9-0-0-2j.zip), add saxon9.jar to your CLASSPATH. There are also nine other files, saxon9-ant.jar, saxon9-dom.jar, saxon9-dom4j.jar, saxon9-jdom.jar, saxon9-s9api.jar, saxon9-sql.jar, saxon9-xom.jar, saxon9-xpath.jar, and saxon9-xqj.jar. These .jar files enable additional functions; see the Saxon documentation for more information about them. For most of what we’ll do in this book, saxon9.jar is all you’ll need.

Once you’ve installed Saxon and updated your classpath, go to a command prompt and type the following command:

```
java net.sf.saxon.Transform
```

You should get a message like this:

```
No source file name
Saxon 9.0.0.3j from saxonb9-0-0-2j.jar
Usage: see http://www.saxonica.com/documentation/using-xsl/commandline.html
```
The error message will list dozens of options. Most of the time we’ll simply specify the 
source XML file and the XSLT stylesheet.

Saxon is also available in a closed source version, Saxon-SA, that provides complete 
support for the XML Schema functions defined in XSLT 2.0, XPath 2.0, and XQuery 
1.0. All of the examples in this book that use schema-aware functions were tested with 
the closed source, commercial version of Saxon.

To install the schema-aware version of Saxon, you need to add saxon9sa.jar to your 
CLASSPATH. When you purchase Saxon-SA, you’ll get a saxon-license.lic file; put that file 
into the Saxon/bin directory and add that directory to your system PATH. The command 
to run the schema-aware version of Saxon is slightly different:

```
java com.saxonica.Transform
```

The version number will be different, but everything else should be the same:

```
No source file name
Saxon-SA 9.0.0.3J from Saxonica
Usage: see http://www.saxonica.com/documentation/using-xsl/commandline.html
Options:
   -a                    Use xml-stylesheet PI, not style-doc argument
...                  
```

## Installing the Microsoft XSLT Processor

The most commonly used XSLT processor in the .NET world is the Microsoft XSLT 
processor. The best way to find the tools is to visit http://msdn.microsoft.com/xml. As 
of this writing (2008), the file you want to download is msxsl.exe. Put this on your 
system path, then go to a command prompt and type the following command:

```
msxsl
```
You’ll see a message like this:

Microsoft (R) XSLT Processor Version 4.0

Usage: MSXSL source stylesheet [options] [param=value...] [xmlns:prefix=uri...]

Options:
-? Show this message
-o filename Write output to named file
-m startMode Start the transform in this mode
-xw Strip non-significant whitespace from source and stylesheet
...

In Chapter 9, we’ll look at C# code that uses the XSLT processor built into the .NET framework. If we’re just transforming XML documents from the command line, msxsl.exe is all we’ll need.

**Installing the Altova XSLT Engine**

As of this writing (early 2008), the only zero-cost XSLT 2.0 processor that provides schema support is the Altova XSLT engine. This is the XSLT processor at the heart of Altova’s XMLSpy product. It is currently a Windows-only download available under a royalty-free license at [http://www.altova.com/altovaxml](http://www.altova.com/altovaxml).

To install the engine, download the software (currently a setup file named altovaxml2008.exe) and run it. At a command prompt, type the following command:

```
altovaxml
```

You’ll see a message like this:

AltovaXML Version 2008 sp1
Copyright (c) 1998-2007 Altova GmbH. All rights reserved.
Use of this software is subject to the license agreement at
http://www.altova.com/altovaxmldla.html

Use the xslt1 engine:
```
xslt1 <filename> /in <filename> [/param name=value] [/out <filename>]
```

Use the xslt2 engine:
```
xslt2 <filename> /in <filename> [/param name=value] [/out <filename>]
```

Use the xquery engine:
```
xquery <filename> [/in <filename>] [/param name=value] [/out <filename>]
[serialization options]
```

Use the validator:
```
/validate <filename> [/schema <filename> | /dtd <filename>]
/wellformed <filename>
```

Parameters:
```
/validate, /v <filename> Schema validates the specified XML file
/wellformed, /w <filename> Check if specified XML file is well-formed
/xslt1 <filename> Sets the source for XSLT1 stylesheet
/xslt2 <filename> Sets the source for XSLT2 stylesheet
/xquery, /xq <filename> Sets the source for XQuery expression
```

---

Installing XSLT Processors | 23
Summary

In this chapter, we’ve gone over the basics of XML and talked about DOM and SAX, two standards that are commonly used by XSLT processors. We also talked about other technology standards and how to install several stylesheet processors. At this point, you’ve got everything you need to build and use your first stylesheets, which we’ll do in the next chapter.
In future chapters, we’ll spend a lot of time talking about XSLT, XPath, and various advanced functions used to transform XML documents. First, though, we’ll go through a short example to illustrate how stylesheets work.

Goals of This Chapter

By the end of this chapter, you should know:

• How to create a basic stylesheet
• How to use a stylesheet to transform an XML document
• How a stylesheet processor uses a stylesheet to transform an XML document
• The structure of an XSLT stylesheet

Transforming Hello World

Continuing the tradition of Hello World examples begun by Brian Kernighan and Dennis Ritchie in *The C Programming Language* (Prentice Hall, 1988), we’ll transform a Hello World XML document.

Our Sample Document

First, we’ll look at our sample document. This simple XML document, courtesy of the XML 1.0 specification, contains the famous friendly greeting to the world:

```xml
<?xml version="1.0"?>
<!-- greeting.xml -->
<greeting>
  Hello, World!
</greeting>
```

What we’d like to do is transform this XML document into something we can view in an ordinary household browser.
A Sample Stylesheet

Here’s an XSLT stylesheet that defines how to transform the XML document:

```xml
<?xml version="1.0"?>
<!-- greeting.xsl -->
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <xsl:apply-templates select="greeting"/>
    </xsl:template>
    <xsl:template match="greeting">
        <html>
            <body>
                <h1>
                    <xsl:value-of select="."/>
                </h1>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

We’ll talk about these elements and what they do in just a minute. Keep in mind that the stylesheet is itself an XML document, so we have to follow all of the document rules we discussed in the previous chapter.

Transforming the XML Document

To transform the XML document using the XSLT stylesheet, run this command if you’re using Xalan:

```
```

For Saxon, the command looks like this:

```
java net.sf.saxon.Transform -o greeting.html greeting.xml greeting.xsl
```

If you’re using the Schema-aware version of Saxon, the name of the Java class is different:

```
java com.saxon.Transform -o greeting.html greeting.xml greeting.xsl
```

The command for the Altova XSLT engine is:

```
altovaxml /xslt1 greeting.xsl /in greeting.xml /out greeting.html
```

Finally, if you’re using Microsoft’s MSXSL, type this command:

```
msxsl greeting.xml greeting.xsl -o greeting.html
```
This command transforms the document greeting.xml, using the templates found in the stylesheet greeting.xsl. The results of the transformation are written to the file greeting.html. Check the output file in your favorite browser to make sure the transformation worked correctly.

This is one of the few times in this book we’ll cover the syntax of the command to run a transformation. The exception to this rule will be when you need to do something more advanced (pass parameters to a stylesheet, for example). Typically, all you need to know are the filenames of the XML, XSL, and output files, and the format of the command for your stylesheet processor.

Stylesheet Results

The XSLT processor generates these results:

```xml
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

When rendered in a browser, our output document looks like Figure 2-1. Congratulations! You’ve now used XSLT to transform an XML document.

How a Stylesheet Is Processed

Now that we’re giddy with the excitement of having transformed an XML document, let’s discuss the stylesheet and how it works. A big part of the XSLT learning curve is figuring out how stylesheets are processed. To make this clear, we’ll go through the steps taken by the stylesheet processor to create the HTML document we want.

Parsing the Stylesheet

Before the XSLT processor can process a stylesheet, it has to read it. Conceptually, it doesn’t matter how the XSLT processor stores the information from your stylesheet.
For our purposes, we’ll just assume that the XSLT processor can magically find anything it needs in our stylesheet. (If you really must know, Xalan uses an optimized table structure to represent the stylesheet; other processors may use that approach or something else.)

Our stylesheet contains three items: an `<xsl:output>` element that specifies HTML as the output format, and two `<xsl:template>` elements that specify how parts of our XML document should be transformed.

**Parsing the Transformee**

Now that the XSLT processor has processed the stylesheet, it needs to read the document it’s supposed to transform. The XSLT processor builds a tree view from the XML source. This tree view is what we’ll keep in mind when we build our stylesheets.

**Lather, Rinse, Repeat**

Finally, we’re ready to begin the actual work of transforming the XML document. The XSLT processor may set some properties based on your stylesheet (in the previous example, it would set its output method to HTML), then it begins processing as follows:

1. Do I have any nodes to process? The nodes to process are represented by the context. Initially, the context is the root of the XML document, but it changes throughout the stylesheet. We’ll talk about the context extensively in the next chapter. (Note: all XSLT processors enjoy being anthropomorphized, so I’ll often refer to them this way.)

While any nodes are in the context, do the following:

2. Get the next node from the context. Do I have any `<xsl:template>`s that match it? (In our example, the next node is the root node, represented in XPath syntax by `/`). There is a template that matches this node—it’s the one that begins `<xsl:template match="/">`.

3. If one or more `<xsl:template>`s match, pick the right one and process it. The right one is the most specific template. For example, `<xsl:template match="/html/body/h1/p">` is more specific than `<xsl:template match="p">`. (See the discussion of `<xsl:template>` in Appendix A for more information.) If no `<xsl:template>`s match, the XSLT processor uses some built-in rules. See the section “Built-in Template Rules” later in this chapter for more information.

Notice that this is a recursive processing model. We process the current node by finding the right `<xsl:template>` for it. That `<xsl:template>` may in turn invoke other `<xsl:templates>`, which invoke `<xsl:templates>` as well. This model takes some getting used to, but it is actually quite elegant once you’re accustomed to it.
If it helps, you can think of the root template (<xsl:template match="/"/>) as the main method in a C, C++, or Java program. No matter how much code you’ve written, everything starts in `main`. Similarly, no matter how many `<xsl:template>`s you’ve defined in your stylesheet, everything starts in `<xsl:template match="/">`.

### Walking Through Our Example

Let’s revisit our example and see how the XSLT processor transforms our document:

1. The XSLT stylesheet is parsed and converted into a tree structure.
2. The XML document is also parsed and converted into a tree structure. Don’t worry too much about what that tree looks like or how it works; for now, just assume that the XSLT processor knows everything that’s in the XML document and the XSLT stylesheet. After the first two steps are done, when we describe various things using XSLT and XPath, the processor knows what we’re talking about.
3. The XSLT processor is now at the root of the XML document. This is the original context.
4. There is an `xsl:template` that matches the document root:
   ```xml
   <xsl:template match="/">
     <xsl:apply-templates select="greeting"/>
   </xsl:template>
   ```
   A single forward slash (`/`) is an **XSLT pattern** (written in `XPath`) that matches “document nodes.”
5. Now the process begins again inside the `xsl:template`. Our only instruction here is to apply whatever `xsl:template`es might apply to any `greeting` elements in the current context. The current context inside this template is defined by the `match` attribute of the `xsl:template` element. This means the XSLT processor is looking for any `greeting` elements at the document root.

Because one `greeting` element is at the document root, the XSLT processor must deal with it. (If more than one element matches in the current context, the XSLT processor deals with each one in the order in which they appear in the document; this is known as **document order**.) Looking at the `greeting` element, the `xsl:template` that applies to it is the second `xsl:template` in our stylesheet:

```xml
<xsl:template match="greeting">
  <html>
    <body>
      <h1>
        <xsl:value-of select="."/>
      </h1>
    </body>
  </html>
</xsl:template>
```
About the Author

Doug Tidwell is a senior programmer at IBM. He has more than a sixth of a century of programming experience, and has been working with markup languages for more than a decade. He was a speaker at the first XML conference in 1997 and has taught XML classes around the world. His job as a cyber evangelist is to look busy and to help people use new technologies to solve problems. Using a pair of zircon-encrusted tweezers, he holds a master’s degree in computer science from Vanderbilt University and a bachelor’s degree in English from the University of Georgia. He lives in Raleigh, North Carolina, with his wife and their daughter Lily.

Colophon

The animal on the cover of XSLT, Second Edition, is a Jabiru (Jabiru mycteria). Standing up to five feet tall and with a wingspan of eight feet, this wading stork is the largest flying bird in the western hemisphere. The bird’s habitat ranges from southern Mexico to northern Argentina, and much of its migrating population is found in Belize from November through July. Its habitat generally includes coastal areas, savannas, and marshes, and it feeds on freshwater wildlife such as frogs, fish, and even snakes. Its plumage is mostly white, but its head, neck, and beak are jet black. A featherless red pouch at the base of its neck gives the Jabiru its name, which means “swollen neck” in the Tupi-Guarani language.

The Jabiru population has steadily decreased over the past decades due to hunting and deforestation, but some areas of Central America have seen a recovery in the bird’s numbers. It is currently considered a species of least concern by the World Conservation Union, an improvement from a status of near-threatened in 1988.

The cover image is an original antique engraving from the 19th century. The cover font is Adobe’s ITC Garamond. The text font is Linotype Birka, the heading font is Adobe Myriad Condensed, and the code font is LucasFont’s TheSansMonoCondensed.