Michiel van Otegem

Teaches

XSLT from the ground up for people with little or no knowledge of it

Apply

your knowledge in the real world

SAMS Teach Yourself XSLT in 21 Days

SAMS

Sams Teach Yourself XSLT In 21 Days

Day 1
Familiarize yourself with XSLT, what it is for and what its place is among the XML technologies.

Day 2
Start building your first XSLT stylesheets and learn about the XSLT basic elements.

Day 3
Select data from an XML source based on the location of that data in the source document.

Day 4
Get to know one of the most important elements in XSLT: the template, which you can use to process and format data.

Day 5
Insert all kinds of text, elements, and attributes into your output, based on values in the source document if necessary.

Day 6
Execute code only if certain conditions are met, and operate on multiple elements with the same code.

Day 7
Make sure that your output is encoded correctly, and create HTML or text instead of XML.

Day 8
Make expressions easier and store information in variables so the data can be accessed easily.

Day 9
Alter the output of a template or an entire stylesheet based on an external value, not from the XML source document.

Day 10
Get a grip on the types of data XSLT has and the intricate properties they have.

Day 11
Check and manipulate strings to represent data types that are not normally available in XSLT.

Day 12
Use sorting and numbering to make long documents easier to read.

Day 13
Include and import functionality from existing stylesheets, so you don't have to reinvent the wheel for each stylesheet.

Day 14
Break up your source document into smaller documents so they are easier to handle.

Day 15
Gracefully separate data with namespaces and select data using a namespace.

Day 16
Select data with complex expressions and make complex expressions easier to cope with.

Day 17
Use recursion to do tasks that you can't do with matching or iteration.

Day 18
Create stylesheets that perform calculations, such as currency conversion and calculating an order.

Day 19
Extend the processor to perform functions that are normally impossible in XSLT.

Day 20
Deal with differences between processors and XSLT versions.

Day 21
Learn how to design XML and XSLT documents effectively, so your applications are easier to create and maintain.

Michiel van Otegem

SAMS Teach Yourself XSLT in 21 Days

SAMS

800 East 96th St., Indianapolis, Indiana, 46240 USA

Sams Teach Yourself XSLT in 21 Days

International Standard Book Number: 0-672-32318-4

Library of Congress Catalog Card Number: 2001094829

Printed in the United States of America

First Printing: February 2002

07 06 05 5 4 3 2

Associate Publisher
Paul Boger

Acquisitions Editor
Rochelle J. Kronzek

Development Editor
Songlin Qiu

Managing Editor
Matt Purcell

Project Editor
Natalie Harris

Copy Editor
Chuck Hutchinson

Indexer
Kelly Castell

Proofreader
Kay Hoskin

Technical Editor
Mike Wooding

Team Coordinator
Pamalee Nelson

Interior Designer
Dan Armstrong

Cover Designer
Alan Clements

Page Layout
Stacey Richwine-DeRome
Gloria Schurick

Contents at a Glance

Introduction 1

WEEK 1 At a Glance 5

Day 1 Getting Started with XSLT 7
2 Transforming Your First XML 35
3 Selecting Data 59
4 Using Templates 83
5 Inserting Text and Elements 107
6 Conditional and Iterative Processing 139
7 Controlling the Output 163

WEEK 1 In Review 191

WEEK 2 At a Glance 205

8 Working with Variables 207
9 Working with Parameters 229
10 Understanding Data Types 249
11 Working with Strings 269
12 Sorting and Numbering 295
13 Working with Multifile Stylesheets 319
14 Working with Multiple XML Sources 341

WEEK 2 In Review: 363

WEEK 3 At a Glance 373

15 Working with Namespaces 375
16 Advanced Data Selection 399
17 Using Recursion 423
18 Building Computational Stylesheets 441
19 Working with XSLT Extensions 465
20 Working with Different Processors 485
21 Designing XML and XSLT Applications 507

WEEK 3 In Review 527

Appendixes

Appendix A Answers to Quiz Questions and Exercises 547

B A Quick Reference of XSLT Elements and Functions 579

C Command-Line Reference for Selected Processors 617

D XML Resources on the Web 623

Index 625

Contents

Introduction 1

Who Is This Book's Intended Audience? 1
What Do You Need to Know Before You Read This Book? 1
What Will You Learn from This Book? 2
What Software Will You Need to Complete the Examples Provided with This Book? 2
How This Book Is Organized 2
What's on the Sams Web Site for This Book 3
Conventions Used in This Book 3

WEEK 1 At a Glance 5

DAY 1 Getting Started with XSLT 7

Overview of XSLT 8
    Introduction to XML and XSLT 9
    XSLT and the XML Family 11
    The Benefits of XSLT 16
    When Not to Use XSLT 21
How Does XSLT Work? 22
    XSLT Transformation Explained 23
    Understanding Declarative Programming 26
Creating XSLT Files 27
    Using a Text Editor 27
    Using an XML Editor 27
    XSLT Editors and Debuggers 28
    XSLT Design Tools 29
Processors for XML Transformation with XSLT 29
    MSXML 29
    Saxon 30
    Xalan 31
Summary 32
Q&A 32
Workshop 33
    Quiz 33
    Exercise 33

DAY 2 Transforming Your First XML 35

Anatomy of a Stylesheet 36
    What Is a Stylesheet? 36
    Basic Stylesheet Elements 36
    Simplified Stylesheet Syntax 48

Applying a Stylesheet to an XML Source 49
Linking a Stylesheet to an XML Source 49
Embedding a Stylesheet in an XML Source 52
Executing a Stylesheet Using Code 54
Summary 55
Q&A 56
Workshop 56
Quiz 56
Exercise 57

Day 3 Selecting Data 59

Understanding the XML Document Tree 59
What Is a Node? 61
What Is a Node-Set? 62
Understanding XPath 63
Selecting Elements 64
Selecting Attributes 70
Beyond the Basics 72
Summary 80
Q&A 80
Workshop 81
Quiz 81
Exercise 81

Day 4 Using Templates 83

Understanding Templates 83
A Closer Look at Templates 84
The Benefit of Using Templates 86
Creating and Using Templates 92
More About Match Templates 92
Using Named Templates 96
Determining Which Template Is Used 98
Different Templates for Different Cases 98
Template Priorities....100
Adding Your Own Priorities 103
Summary 104
Q&A 105
Workshop 105
Quiz 106
Exercises 106

Day 5 Inserting Text and Elements 107

Inserting Text 108
Text with Special Characters....108
Special Characters in XSLT 111

Inserting Elements and Attributes....113
Inserting Elements....114
Inserting Attributes....123
Copying Elements from the Source Document....129
Copying Only the Context Node....129
Copying Node-Sets and Tree Fragments....131
Inserting Comments and Processing Instructions....133
Inserting Comments....133
Inserting Processing Instructions....135
Summary....136
Q&A....137
Workshop....137
Quiz....137
Exercise....137

DAY 6 Conditional and Iterative Processing 139

Iterating Through a Node-Set....140
Processing Each Node in a Node-Set....140
Filtering Node-Sets....144
Using Node-Set Functions....146
Conditional Processing....148
Simple Conditional Processing....148
Conditional Processing with Multiple Options....152
More About Expressions....154
Formalizing Expressions....154
Using Multiple Predicates....156
Combining Expressions....156
Using Boolean Functions....157
Negating an Expression Result....158
Conversion to Boolean Values....159
Summary....160
Q&A....160
Workshop....161
Quiz....161
Exercises....161

DAY 7 Controlling the Output 163

Creating Different Output Formats....164
Understanding XML Output....164
Creating HTML Output....167
Creating Other Types of Output....171
Specifying the Media Type....174

Output Encoding and Output Escaping 174
Determining the Output Encoding 174
Encoding Characters That Are Not Supported 175
Disabling Output Escaping 176
Controlling Whitespace 179
Understanding Whitespace 179
Stripping Whitespace from the Source Document 180
Dealing with Whitespace in the Stylesheet 185
Indenting 188
Summary 189
Q&A 189
Workshop 190
Quiz 190
Exercise 190

WEEK 1 In Review 191

Overview of Bonus Project 1 191
Creating an Article with a Table of Contents 191
Project Overview 192
Creating the Article XML 193
Creating the XSLT Document 195

WEEK 2 At a Glance 205

DAY 8 Working with Variables 207

Understanding Variables 208
What Are Variables? 208
What Is the Benefit of Variables? 209
Creating and Using Variables 210
Using Simple Variables 210
Using Complex Variables 215
Creating Variables from Expressions 216
Using Variables to Replace an Expression 217
Using Variables for Out-of-Context Data 220
Creating Variables from XSLT Elements 225
Summary 226
Q&A 226
Workshop 227
Quiz 227
Exercise 227

DAY 9 Working with Parameters 229

Understanding Parameters 230
What Are Parameters? 230
What Is the Benefit of Parameters? 231
Using Parameters 232
Using Parameters to Alter the Output 232
Using Parameters to Create Template Functions 236
Getting Data from Outside the Stylesheet 240
Summary 247
Q&A 247
Workshop 248
Quiz 248
Exercise 248

DAY 10 Understanding Data Types 249

Data Type Basics 250
What Is a Data Type? 250
Data Types in XSLT 251
Conversion Between Data Types 259
Explicit Data Type Conversion 259
Implicit Data Type Conversion 262
Conversion Pitfalls 263
Comparing Values 263
Comparison Pitfalls 264
Summary 265
Q&A 266
Workshop 266
Quiz 266
Exercise 267

DAY 11 Working with Strings 269

Operations on Strings 270
Gluing Strings Together 270
Checking for Characters in a String 272
Getting the Length of a String 277
Working with Partial Strings 279
Replacing Parts of a String 283
Formatting Data 285
Formatting Numbers 285
Formatting Date And Time 290
Formatting Other Data 292
Summary 292
Q&A 293

Workshop 293
Quiz 293
Exercises 294

Day 12 Sorting and Numbering 295

Sorting....296
Using Static Sorting....296
Using Dynamic Sorting....304
Numbering 306
Inserting Numbering 306
Controlling the Numbering Output 314
Summary....317
Q&A 317
Workshop 318
Quiz 318
Exercises 318

Day 13 Working with Multifile Stylesheets 319

Using Multiple Stylesheets....320
The Benefits of Multiple Files 320
The Drawbacks of Multiple Files 320
Including Stylesheets 321
Duplicate Templates 328
Duplicate Variables and Parameters 329
Importing Stylesheets 329
The Difference Between Including and Importing 329
How to Import a Stylesheet....330
Overriding Templates 332
Import Rules for Other Elements 335
Summary....339
Q&A 339
Workshop 340
Quiz 340
Exercise 340

Day 14 Working with Multiple XML Sources 341

Accessing Other XML Sources 342
Getting Data from an XML Source....342
Defining Additional Documents Dynamically 353
Linking Source Documents....355
Specifying a Different File Location....357
Accessing the Stylesheet Elements....357
Multidocument Pros and Cons 359

Multidocument Pitfalls 359
Multidocument Best Practices 360
Summary 360
Q&A 361
Workshop 362
Quiz 362
Exercises 362

WEEK 2 In Review 363

Overview of Bonus Project 2 363
Creating a Multifile Stylesheet with Parameters 363
Starting Your Project 364
Creating the Second Stylesheet 369
Creating a Stylesheet for All the Data 371

WEEK 3 At a Glance 373

DAY 15 Working with Namespaces 375

Understanding Namespaces 376
Namespaces Explained 376
The Benefits of Namespaces 380
The Drawbacks of Namespaces 381
Namespaces, DTDs, and Schemas 382
Processing XML Sources with Namespaces 384
Getting Namespace Information 386
Inserting and Removing Namespaces 387
Inserting Nodes with Namespaces 387
Changing Namespaces 389
Removing Namespaces 392
Summary 396
Q&A 397
Workshop 397
Quiz 397
Exercise 398

DAY 16 Advanced Data Selection 399

More About Expressions 400
Matching and Selecting Data 401
Comparing Values 401
Selecting Distinct Values 402
Working with Keys 404
What Is a Key? 404
Using Keys to Select Data 405

Working with Unique IDs 413
Selecting Data with a Unique ID 414
Inserting Unique IDs 415
Using Keys and Generated IDs to Select Distinct Values 418
Summary 419
Q&A 420
Workshop 420
Quiz 420
Exercise 421

Day 17 Using Recursion 423

Understanding Recursion 424
What Is Recursion? 424
Why and When Should You Use Recursion? 425
The Drawbacks of Recursion 427
Creating Recursive Templates 428
Recursion with Single Values 432
Totaling with Recursion 434
Summary 437
Q&A 437
Workshop 438
Quiz 438
Exercises 438

Day 18 Building Computational Stylesheets 441

Computational Stylesheets Explained 442
What Is a Computational Stylesheet? 442
When Do I Use a Computational Stylesheet? 443
Operators and Functions Used in Computations 443
Operators 444
Functions 446
Computational Applications 451
Ranking Teams in a Competition 451
String Manipulation 458
Converting Currencies 460
Summary 462
Q&A 462
Workshop 463
Quiz 463
Exercise 463

Day 19 Working with XSLT Extensions 465

Understanding XSLT Extensions 466
What Are XSLT Extensions? 466
What Are the Benefits of XSLT Extensions? 466

The Drawbacks of XSLT Extensions 467
How Do XSLT Extensions Work? 468
Using Built-in Extensions 468
Using Extension Elements 469
Using Extension Functions 473
Using Extensions with Other Processors 475
Creating Your Own Extension Functions 475
Using Java Functions as Extension Functions 476
Creating an Extension Function with Script 478
Summary 482
Q&A 483
Workshop 484
Quiz 484
Exercises 484

DAY 20 Working with Different Processors 485

Targeting Multiple Processors 486
Key Processor Differences 487
Dealing with Processor Differences 490
Dealing with Different XSLT Versions 495
XML Capabilities of Database Servers 497
Getting and Transforming XML Data with Oracle 498
Getting XML Data from Microsoft SQL Server 502
Summary 503
Q&A 504
Workshop 504
Quiz 505
Exercise 505

DAY 21 Designing XML and XSLT Applications 507

Designing XML 508
XML Design Considerations 510
Design Tools 518
Designing XSLT 519
XSLT Design Considerations 519
XSLT Design Do’s and Don’ts 524
Summary 525
Q&A 525
Workshop 526
Quiz 526
Exercise 526

WEEK 3 In Review 527

Overview of Bonus Project 3 527
Creating a Shopping Basket in XSLT....................528
The Product Data....................528
The Shopping Basket....................529
Displaying the Data....................530
Updating the Basket 536
Invoking the Processor 539

Appendixes

A Answers to Quiz Questions and Exercises 547

Answers for Day 1 547
Answers to Quiz Questions....................547
Solution to Exercise....................548
Answers for Day 2 548
Answers to Quiz Questions....................548
Solution to Exercise....................548
Answers for Day 3 549
Answers to Quiz Questions....................549
Solution to Exercise....................549
Answers for Day 4 550
Answers to Quiz Questions....................550
Solutions to Exercises 550
Answers for Day 5 552
Answers to Quiz Questions....................552
Solution to Exercise....................552
Answers for Day 6 553
Answers to Quiz Questions....................553
Solutions to Exercises 553
Answers for Day 7 554
Answers to Quiz Questions....................554
Solution to Exercise....................555
Answers for Day 8 555
Answers to Quiz Questions....................555
Solution to Exercise....................556
Answers for Day 9 557
Answers to Quiz Questions....................557
Solution to Exercise....................558
Answers for Day 10 559
Answers to Quiz Questions....................559
Solution to Exercise....................559

Answers for Day 11 560
Answers to Quiz Questions 560
Solutions to Exercises 560
Answers for Day 12 563
Answers to Quiz Questions 563
Solutions to Exercises 563
Answers for Day 13 564
Answers to Quiz Questions 564
Solution to Exercise 565
Answers for Day 14 565
Answers to Quiz Questions 565
Solutions to Exercises 565
Answers for Day 15 567
Answers to Quiz Questions 567
Solution to Exercise 567
Answers for Day 16 568
Answers to Quiz Questions 568
Solution to Exercise 568
Answers for Day 17 569
Answers to Quiz Questions 569
Solutions to Exercises 570
Answers for Day 18 571
Answers to Quiz Questions 571
Solution to Exercise 572
Answers for Day 19 573
Answers to Quiz Questions 573
Solutions to Exercises 574
Answers for Day 20 574
Answers to Quiz Questions 574
Solution to Exercise 575
Answers for Day 21 576
Answers to Quiz Questions 576
Solution to Exercise 577

B A Quick Reference of XSLT Elements and Functions 579

XSLT Element Reference 580
`xsl:apply-imports` 580
`xsl:apply-templates` 581
`xsl:attribute` 581
`xsl:attribute-set` 582
`xsl:call-template` 582
`xsl:choose` 583
`xsl:comment` 583

`xsl:copy` 583
`xsl:copy-of` 584
`xsl:decimal-format` 584
`xsl:element` 585
`xsl:fallback` 586
`xsl:for-each` 587
`xsl:if` 587
`xsl:import` 587
`xsl:include` 588
`xsl:key` 589
`xsl:message` 589
`xsl:namespace-alias` 590
`xsl:number` 590
`xsl:otherwise` 591
`xsl:output` 592
`xsl:param` 593
`xsl:preserve-space` 594
`xsl:processing-instruction` 594
`xsl:sort` 595
`xsl:strip-space` 596
`xsl:stylesheet` 596
`xsl:template` 597
`xsl:text` 598
`xsl:transform` 598
`xsl:value-of` 598
`xsl:variable` 599
`xsl:when` 599
`xsl:with-param` 600

XSLT and XPath Function Reference 600
`boolean()` 600
`ceiling()` 601
`concat()` 601
`contains()` 602
`count()` 602
`current()` 603
`document()` 603
`element-available()` 603
`false()` 604
`floor()` 604
`format-number()` 605
`function-available()` 605
`generate-id()` 605

`id()` 606
`key()` 606
`lang()` 607
`last()` 607
`local-name()` 607
`name()` 608
`namespace-uri()` 608
`normalize-space()` 609
`not()` 609
`number()` 609
`position()` 610
`round()` 610
`starts-with()` 611
`string()` 611
`string-length()` 612
`substring()` 612
`substring-after()` 613
`substring-before()` 613
`sum()` 614
`system-property()` 614
`translate()` 614
`true()` 615
`unparsed-entity-uri()` 615

C Command-Line Reference for Selected Processors 617

MSXSL 617
Usage 618
Options 618
Saxon 619
Usage 619
Options 619
Xalan 620
Usage 621
Options 621

D XML Resources on the Web 623

Index 625

About the Author

Michiel van Otegem lives and works in the Netherlands. He is the co-founder and Chief Web development Teacher of ASPNL, a consulting and teaching firm targeting the Dutch and European market. He teaches advanced ASP, ASP.NET, and XML/XSLT classes, and writes articles and tutorials for magazines and Web sites, such as ASPNL.com, TopXML.com, ASPAlliance.com, *CoDe Magazine*, and *asp.netPRO* magazine. He has had a passion for programming ever since he wrote his first programs in MSX Basic and Z80 assembler, at age 10. Now, nearly two decades later, he is a pioneer in Web development, quick to embrace technologies such as XML and ASP(.NET). He has worked with a wide range of languages and platforms, including ASP(.NET), Visual Basic, Access/SQL Server, C/C++, CGI/Perl, PHP, and, of course, XML and XSLT. He is a long-time contributing member of ASPFriends.com mailing lists, which he now helps to moderate and improve as a valued ASP Ace member.

Dedication

*To the love of my life, Annette, for her patience, understanding,
and support, and to my parents for their unconditional support and wisdom.*

Acknowledgments

This book would not have been possible if my partner, Annette, had not run the house during my mental absence needed to write this book. It would also never have been possible if my parents had not told me to follow my dreams, not to mention that they taught me most of what I know about writing and teaching.

I would have never come into a position to write this book if it weren't for my valuable friends Andreas Kviby and Charles Carroll, who showed me that I could do more than development work alone. In addition, the encouragement and insights of Peter Vogel (and his lovely wife), Kurt Cagle, Michael Corning, and many other speakers and writers made this work interesting and fun.

I would also like to thank the people at Sams Publishing, specifically Songlin Qiu for dealing with all my issues, Natalie Harris for her quickness, Chuck Hutchinson for his fabulous copy edits, and Shelley Kronzek for her belief in me as an author. A special thanks to Mike Wooding, whose technical comments and humor were of enormous value.

Finally, I would like to thank Steely Dan for creating great music that helped me during the writing process.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Associate Publisher for Sams, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4770
E-mail:	`feedback@samspublishing.com`
Mail:	Paul Boger Sams 201 West 103rd Street Indianapolis, IN 46290 USA

Introduction

XML is one of the biggest things to hit the World Wide Web since the invention of the Web itself. It has the simplicity of HTML, looking much like it, but at the same time it is much more powerful. This power comes from its generic nature, which makes XML useful for a myriad of applications, not just on the Web, but in any (distributed) computing environment.

If you want to manipulate XML, you have several choices. By far the most powerful is XSLT, which enables you to do very powerful things with the data stored in an XML document. What makes XSLT so interesting is that it is remarkably simple, but at the same time very powerful. Operations that require many lines of code with conventional techniques can be solved in XSLT with just a few lines of code because XSLT uses a completely different programming paradigm, one that you'll learn to love during the course of reading this book.

XSLT doesn't replace existing programming languages, but rather complements them. In that sense, XSLT is just another tool in your toolbox. However, applications that target XML- and XSLT-enabled Web browsers, such as Internet Explorer, don't need any additional programming. This means you can create distributed applications with little effort.

Finally, because XML and XSLT are World Wide Web Consortium–endorsed standards, they are truly cross-platform. Any platform equipped with an XSLT processor can run your application.

Who Is This Book's Intended Audience?

This book is intended to teach absolute beginners the basics of XSLT and much, much more. This means that this book is also suitable for people with basic knowledge and experience with XSLT, because many of the topics are covered in great detail. In addition, the more advanced topics haven't been forgotten.

What Do You Need to Know Before You Read This Book?

This book starts at the very beginning of XSLT, so you don't need any prior knowledge of XSLT. Because XSLT operates on and is itself XML, you need a basic knowledge of XML. This means you need to know what XML is, what its syntax is, and how it is structured. Beyond that, you really don't need anything else. Having a working knowledge of HTML does help, however. Any prior programming experience is not required;

in fact, XSLT programming is based on another programming paradigm than that used in languages such as C++, Java, and Visual Basic. Any prior programming experience is therefore of limited use.

What Will You Learn from This Book?

This book will teach you anything you need to know about XSLT as a programming language. You will learn how to create XSLT documents and how to use them to transform XML documents to text, HTML, or other XML formats. You also will learn how to use processors to apply XSLT to XML documents and how to use data that is not in the XML document. After you finish the book, you will be able to create complex XSLT documents performing complex transformations of XML documents.

What Software Will You Need to Complete the Examples Provided with This Book?

To complete the samples in the book, you need a text editor and an XSLT processor. Unless otherwise specified, the processor used in this book is Saxon version 6.2.2. Other processors discussed are Xalan Java and Microsoft's MSXML parser/processor component, complemented by the MSXSL command-line tool for this component. You can find information about these processors, including download locations, in Appendix C. At `http://xml.startkabel.nl/`, you can find links to most processors available, most of which are free.

How This Book Is Organized

This book is organized so you learn XSLT in 21 days. Therefore, there are 21 lessons, one lesson for each of the 21 days. The lessons are grouped in equal parts of seven lessons, so one part corresponds to one week. Each week concludes with a Bonus Project, which creates an application from scratch, based on the topics covered in that week. This book also contains several appendixes.

Week 1 aims to build your basic knowledge of XSLT. You will learn about processors, editors, and most importantly, about the structure and elements in XSLT. After completing week 1, you will have a good working knowledge to create basic XSLT documents.

Week 2 extends your knowledge of XSLT, based on what you learned in week 1. In this week, you will learn about the more intricate details of XSLT and how to create more complex and flexible documents. You also will learn how to create applications that span multiple documents.

Week 3 discusses a myriad of different topics that go beyond day-to-day use of XSLT, such as performing computations with XSLT and using processor-specific constructs. The last day also looks back at all that you have learned from an application design point of view, which will help you to design and implement your applications to be flexible and more robust.

This book includes four appendixes:

- Appendix A contains the answers to the questions and exercises in the book.
- Appendix B contains a quick reference to all elements and functions in XSLT.
- Appendix C contains information and a command-line reference on MSXSL, Saxon, and Xalan Java.
- Appendix D contains a list of useful XML and XSLT resources on the Web.

What's on the Sams Web Site for This Book

The chapter-by-chapter code files described in this book are available on the Sams Web site at `http://www.samspublishing.com/`. Enter this book's ISBN in the Search box and click Search. When the book's title is displayed, click the title to go to a page where you can download all the code in a chapter-by-chapter zip file format.

Conventions Used in This Book

The following typographic conventions are used in this book:

- Code lines, commands, statements, variables, and any text you type or see onscreen appear in a `mono` typeface. **`Bold mono`** typeface is used to represent the user's input.
- Placeholders in syntax descriptions appear in an *`italic mono`* typeface. Replace the placeholder with the actual filename, parameter, or whatever element it represents.

NEW TERM
- *Italics* highlight technical terms when they're being defined. A paragraph that defines technical terms is marked by an icon.
- The ➥ icon is used before a line of code that is really a continuation of the preceding line. Sometimes a line of code is too long to fit as a single line on the page. If you see ➥ before a line of code, remember that it's part of the line immediately above it.

OUTPUT
- Code listings that show output are marked with an output icon.
- The book also contains Notes, Tips, and Cautions to help you spot important or useful information more quickly. Some of them are helpful shortcuts to help you work more efficiently.

WEEK 1

At a Glance

XSLT is often regarded as a complex technology. In fact, it is relatively simple, with fewer commands than most other programming languages. The problem is that XSLT requires you to think differently. Instead of requiring step-by-step task-oriented thinking, XSLT requires pattern-oriented thinking, which takes getting used to. When you get used to this way of thinking, XSLT is not very hard to work with and probably easier to grasp for non-programmers such as HTML developers.

In this first week, the focus is on familiarizing you with the way of thinking required for XSLT. You will learn about the differences with other languages, but also the similarities.

Day 1, “Getting Started with XSLT,” will kick off this discussion by giving you an idea what XSLT is for, what its place is among other technologies, and how you can create and run XSLT documents.

On Day 2, “Transforming Your First XML,” it’s time to get your feet wet and do some programming. Day 2 focuses on becoming familiar with the basic building blocks of XSLT. It also will show you nearly every conceivable way of applying XSLT documents to XML documents.

Day 3, “Selecting Data,” is key to using XSLT. XML, and thus XSLT, is all about data. To manipulate data, you need to be able to select the data that you want to manipulate. The focus is therefore on understanding the structure of XML documents and how you can use that structure to select exactly the data you need.

Day 4, “Using Templates,” builds on Day 3 and is equally important to using XSLT. Templates are the basic units of functionality that XSLT offers. When used properly, they can make an XSLT programmer's life much easier. The focus of Day 4 is therefore on giving you a firm grasp on what templates are and how to benefit from them.

After Day 4, you will have all basic tools to transform an XML document and create output. Day 5, "Inserting Text and Elements," looks at how to use these tools to create output that contains more than just text. You will learn how to create a new XML document from another by using XSLT, which is actually the basic purpose of XSLT.

On Day 6, "Conditional and Iterative Processing," you will learn to make choices based on the data in an XML document. This lesson will also revisit some of the topics of Day 3, elaborating more on how to select data based on certain conditions. Day 6 also discusses another form of processing that is more familiar to traditional programmers.

To finish off the week, Day 7, "Controlling the Output," looks at creating different forms of output. Besides XML, you will learn how to create HTML and text-based output. Binary output formats such as PDF and RTF will also be discussed.

After you finish Week 1, you will have a firm grasp of the basics of XSLT. You will be able to transform XML into different outputs for different purposes and with different looks. This knowledge is already enough for you to perform most of the tasks that XSLT was designed for, so this is a big week!

DAY 1

Getting Started with XSLT

With more and more people using Extensible Markup Language (XML) in their applications, the need arises for a generic language to manipulate XML documents. Extensible Stylesheet Language Transformations (XSLT) is this language. It was developed by the W3 Consortium (W3C) and now has Recommendation status, the closest you can get to a standard on the World Wide Web.

Because XML documents themselves don't contain any formatting information, you need something else to format and display the data so that it looks pleasant. With XSLT, you have a language to manipulate an XML document. From that document, you then can create another document that contains formatting information, such as Hypertext Markup Language (HTML), Portable Document Format (PDF), or Rich Text Format (RTF). In addition, you can use XSLT to restructure XML documents.

Today you will learn the following:

- What XSLT is
- What the benefits of XSLT are
- How XSLT performs transformations

- Which tools to use to create XSLT documents
- How to use processors to perform transformations

Overview of XSLT

In the past few years, the World Wide Web has exploded in size. You might find that the sheer number of Web sites and pages is incomprehensible. Because these pages contain only formatting information and almost no information regarding their content, finding the information you need becomes harder as the Web grows. Pick a search engine on the Web and enter a topic. Chances are you'll get thousands of results, most of them irrelevant. In addition to this problem, nearly all information on the Web is encoded in HTML, which was specifically designed to format text for display in browsers. Using this format is fine if you're viewing pages with a browser, but applications designed to process information have a lot of trouble working with HTML because the data in an HTML document has no meaning, and HTML is too unstructured to easily retrieve the data stored in it.

Enter the Extensible Markup Language, or XML. Although much like HTML, it doesn't contain formatting information; instead, it contains information about the meaning of data in a document. This effectively means that any document written in XML provides the meaning of its data as part of the document.

Note

Throughout this book, I will use the term *XML document* for both XML files and XML stored in other forms, such as a string in memory or in a database.

The obvious benefit is that this XML data can be extracted and matched more closely to a search query. The query thus displays the information you actually want and discards everything that is irrelevant. A search engine using XML information can also ask you to refine your query—for instance, to determine whether you meant *computer chips* or *French fries* when you entered the search word *chips*. The idea is that eventually the Web will change from pages of text into the *semantic Web*, where all pages have meaning, not only to people but also to applications.

Although XML is oriented toward the Web and initiated by the W3C, it is not meant for use on the Web alone. It can be used in all sorts of applications, both for storing data or as a means of communication between applications. This usage might seem a little odd, as there has always been a distinction between storing and communicating data. However, it is actually quite natural: Data is data, no matter where you use it. This concept is gradually gaining ground, as more and more vendors use XML in their applications. Microsoft, for instance, now has XML support in most of its products, in one way or another. In fact, the .NET Framework, which has been developed to run most future applications and services, is more or less built around XML. Other vendors embracing XML include Sun and IBM, using it in various applications.

Introduction to XML and XSLT

An XML document looks a lot like an HTML document. Like HTML, XML uses tags that have bearing on what is inside them, as in this example:

```
<title>Teach Yourself XSLT in 21 Days</title>
```

A major advantage is that XML is just text, not some proprietary or binary data format. This means that you can read and edit it with a text editor; an added advantage is that any computer can read it and retrieve data from the document. The latter advantage is made possible by the fact that the tags around the data tell the computer the meaning of that data. The downside is that formatting information is no longer associated with the data, so displaying the data nicely for a human reader is not possible when only XML is used. You might be able to understand and edit XML, but it doesn't look nice, and it certainly isn't displayed in a manner appropriate for the purpose you are using it for. Consider this book, for example. If each paragraph, header, and so on were tagged with XML, reading it that way would be much harder than reading it the way it's formatted now. So, to format the data appropriately, you need to manipulate it before it can be displayed.

What Is XSLT?

If you didn't have a generic tool or language to manipulate XML, formatting it for display would be very hard. You would have to write your own application to read XML and display it in the way you want. You would have to tell your application how to format each different XML tag. So, what if you wanted to change the formatting? You would have to start all over. To remedy this problem, the W3C started development of the Extensible Stylesheet Language (XSL), which is a generic language to manipulate and display data in an XML document.

NEW TERM XSL consists of XSL Formatting Objects (XSLFO) and XSL Transformations (better known as XSLT). The former, officially still called XSL, is an XML vocabulary that defines elements used to specify how an XML document is to be displayed. An *XML vocabulary* is a set of XML tags that have been defined for a certain purpose. XHTML is another example of an XML vocabulary.

NEW TERM *XSLT* is a language used to manipulate XML structures or documents. It is also an XML vocabulary. The actual manipulation of an XML document with XSLT is called transformation. *Transformation* is the process of creating a new document based on the original document. This process does not change the source document.

XSLT is extremely versatile and can be used to convert XML to many other forms. All transformations result in a new tree structure. XSL can even be used to create XSLFO documents, which are useful for creating documents that native applications can act upon. XSLFO documents can be used to create Adobe PDF or Microsoft Word files, for example. The main idea here is that XSLFO is generic and can be used for formatting on different "surfaces," as it were.

Note XSLFO has many capabilities for high-quality formatting; this topic is beyond the scope of this book, which covers only XSLT.

What Does XSLT Do?

XSLT transforms an XML document into another document, which can contain XSLFO tags to format the document's data for display, but this is not required. In fact, you are not required to use XSLT as part of XSL at all. Like XSL is designed for use by many applications, XSLT is designed to transform to many different outputs. So, like XSL, XSLT is a generic language to be used by many applications across many platforms. With XSLT, you can create HTML, XHTML, plain text, PDF, and a number of other document types. You also can use XSLT to transform an XML document into another XML document with a different structure. You may not see the benefit of this capability just now because you can simply use the data from the original document. You will find, however, that transforming into a different XML document is actually very powerful and useful for many applications.

What Does XSLT Look Like?

XSLT is a programming language that transforms XML documents; it, however, is unlike other languages. It differs in look, style, and operation. XSLT is itself XML, and like XSLFO, it is an XML vocabulary. However, its tags don't tell a program how to display something but rather what to do when it encounters a certain tag. If you have programmed before, some of these tags have familiar names and functions. Other tags will look totally unfamiliar, as you can see in Listing 1.1.

LISTING 1.1 XSLT Sample

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8"
    omit-xml-declaration="yes" indent="yes" />
  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match="books">
    <xsl:for-each select="book">
      <xsl:value select="title" />
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Note

Understanding the function of Listing 1.1 is not important right now. This sample merely shows what XSLT looks like.

Note

If you use Internet Explorer 5.0 or higher, you can type in the preceding code and view it. You will see that a stylesheet is XML and has a tree structure.

ANALYSIS Each tag in XSLT expresses a command to the program performing the transformation. In XSLT, unlike most programming languages, these commands may span multiple lines. In fact, because XSLT is itself XML, it has to conform to the same syntax rules as any other XML document. If you have programming experience with Visual Basic, C++, Java, and so on, this coding may look rather strange because there is no concept of lines performing a certain action. XSLT actually works differently from these languages. Don't worry about this point for now; I'll discuss it in more detail later.

XSLT and the XML Family

XSLT relies on and interacts with many of the other members of the XML family. Knowledge of XSLT's place in the XML family and where it came from will help you better understand XSLT itself and how it can be used. Because the XML family has become rather large and is constantly evolving, the following sections will not provide a roadmap to all XML family members.

A Brief History of XML and XSLT

I won't bore you with a long and detailed history of XML and XSLT. I'll just give a brief history so that you can put them in perspective. After all, they didn't just pop up out of thin air. XML is actually based on concepts that were developed in the early days of computing.

The Roots of XML

XML is based on the Standard Generalized Markup Language (SGML), but SGML is much more complex. In that sense, XML is based more on HTML because the design goal for XML was to be as general as SGML but as easy as HTML so that it would be adopted easily. SGML was actually developed as Generalized Markup Language (GML) in 1969 by Ed Mosher, Ray Lorie, and Charles F. Goldfarb at IBM Research. Although the International Standards Organization (ISO) adopted SGML as a data storage and exchange standard in 1986, it was far too complex for widespread recognition. HTML, on the other hand, is the most popular markup language in existence, mostly because of its simplicity.

New Term XML is actually a simplified subset of SGML and as such is much easier to understand and program with. This ease of use also makes it much easier to create a parser for XML than for SGML, which can account for a lot of XML's popularity. A *parser* is a program that can read and understand the syntax and grammar of a language.

Two years after its inception, in February 1998, XML 1.0 became a W3C Recommendation. Since then, work on XML-based languages and systems has taken flight. XHTML, WML, SVG, XPath, XPointer, and XML Query are just some of the XML technologies that have been developed. Some are already W3C Recommendations, whereas others are still under development. XSLT is also one such language, and it became a W3C Recomendation on November 16, 1999.

The Roots of XSLT

Like XML, XSLT is also based on existing concepts. In the early 1990s, an SGML-based standard called Document Style Semantics and Specification Language (DSSSL) was created. This generalized language was meant to be used to manipulate and transform SGML documents to a form that could be displayed or printed. However, because this technology was so complex and hard to use (and thus expensive), only some large publishing houses could afford to create applications for it, for use in high-quality typesetting. SGML and DSSSL approaches are still in use, and as long as no XML/XSLT applications are developed to replace them, they will remain in use.

XSLT has its roots in DSSSL but is much simpler. Although XSLT is based on DSSSL, it is not a real subset of DSSSL, as some XSLT features do not exist in DSSSL.

Other Members of the XML Family

Besides XML and XSL, there are quite a few other members of the XML family—some specific to XML, others shared with technologies such as HTML.

Defining XML Structures

If you want to define an XML structure or vocabulary, you have two options. You can use either a Document Type Definition (DTD) or the more recent XML Schema. The benefit of defining such a structure is that all the documents that conform to the definition will have a predefined structure, making it easier to write applications that use these documents. DTD or XML Schema definitions can be placed in the XML file itself (internal) or in an external file (external). In the latter case, the XML document has to reference the DTD or XML Schema file.

There are two types of parsers:

- Validating parsers
- Nonvalidating parsers

A validating parser raises an error if an XML document does not conform to the rules in the associated DTD or Schema; a nonvalidating parser does not.

Note

Most validating parsers have the option to turn off validation, effectively making them nonvalidating parsers.

A DTD itself is not XML and as such cannot be used by a parser for tasks other than validating an XML document. The DTD cannot be queried as XML. Some parsers offer functions to get information from the DTD, but they are separated from functions that apply to the XML. Also, DTDs provide little information about the data type of values. Only a few data types are known in DTDs, but most applications have many more. You cannot define new data types, so you are stuck with the data types that DTD offers.

Each XML document can have only a single DTD that corresponds to it. Complex DTDs can be created to aggregate other DTDs, but the XML document itself is associated with a single DTD.

To remedy most of the problems with DTDs, the W3C created XML Schemas. XML Schemas, which received Recommendation status on May 2, 2001, are XML and also much more flexible than DTDs. They also define more data types than DTDs do. Many developers were quick to adopt XML Schemas, even before they became a W3C Recommendation.

Telling Apart Vocabularies

By using XML Schemas (or DTDs), you can define XML vocabularies. If you mix vocabularies, however, you may have vocabularies that have elements with the same names. This problem can't be solved by defining vocabularies alone. XML namespaces have been added to the XML family to solve this problem. XML namespaces are designed to keep XML vocabularies apart. For each vocabulary used in a document, you can define a different namespace with a unique name within the document. This namespace definition can point to an existing XML Schema or DTD, but this is not required. The benefit of pointing the namespace to an actual DTD or XML Schema is that everybody knows what the document is supposed to look like.

You can identify a namespace in an XML document because the namespace appears in front of the element or attribute name. The namespace and element or attribute are separated by a colon. Say you define a namespace for books. A title element using the book namespace would look like this:

```
<book:title>Teach Yourself XSLT in 21 Days</book:title>
```

You may remember the XSLT sample in Listing 1.1. In that sample, all elements that were part of XSLT were preceded by the namespace `xsl`. XSLT itself uses an XML namespace to work. A namespace for a document is defined as an attribute of an element—in most cases, the root element. For XSLT, the namespace is defined as follows:

```
xmlns:xsl=http://www.w3.org/1999/XSL/Transform
```

The namespace definition has three parts. The attribute name has two parts: the namespace declaration `xmlns` (a predefined namespace), followed by the namespace you want to introduce (in this case, `xsl`).

Note

The introduced namespace does not necessarily have to be the same in every document. The `xsl` namespace in XSLT is just a convention. If you like, you can create another namespace (for example, `transform`) and use it instead of `xsl`.

NEW TERM The last piece of the declaration is a Uniform Resource Identifier (URI), which must be unique for each vocabulary. A *URI* is a unique name or address for a resource. It can be a Uniform Resource Locator (URL) or a Uniform Resource Name (URN).

The URI could point to a DTD or XML Schema to define the vocabulary that the namespace represents, but this is not required. Also, if no validation is involved, the namespace can contain any element name. The namespace declaration at the beginning of this section specifically points to a URI that defines it as being XSLT. If you use another namespace for XSLT, the only requirement is that it points to the same URI.

Note

You will learn more details about XML namespaces on Day 15, "Working with Namespaces."

The Document Object Model

If you want to program with XML, you obviously need a way to get to the data. You could write a program that reads the XML from a file and does something with it. This approach would probably yield a proprietary solution. The whole idea behind XML is that it is a standard anybody can use, so creating a proprietary solution isn't the way to go. What you need is a standard Application Programming Interface (API) to interact with the XML. The W3C created the Document Object Model (DOM) to serve as a standard API. The idea behind DOM is that an XML document is a hierarchical tree of elements and attributes that can be represented in memory and manipulated through a standard mechanism. DOM is not limited to use with XML; it also works with HTML 4.0. Because HTML is less structured than XML, though, you might run into some problems using it with HTML.

Like the other members of the XML family, DOM is evolving. Since November 13, 2000, DOM Level 2 has Recommendation status. DOM Level 3 is currently under development.

Simple API for XML

DOM is a standard by decree of the W3 Consortium. However, before DOM was around, people were in need of a standard method to use XML. The method that emerged to be the de facto standard, Simple API for XML (SAX),takes an entirely different route than DOM when working with an XML document. Instead of reading the entire document, SAX reads a document an element at a time. Each element that the parser encounters will fire an event. You can attach a routine to the event to act on the element and generate output, if you like.

A big advantage to this approach is that hardly any memory is involved because you don't need to build the entire document in memory. This approach is extremely useful when you're working with large documents.

Many parsers use the SAX model. Some others use DOM or, like Microsoft's MSXML parser, offer a choice between the two.

Addressing Elements and Attributes

Because XML documents have a hierarchical tree structure, you can address elements through a path expression, which is similar to a path expression addressing a file on a file system or in a Web site. With this analogy, you can compare elements of an XML document with folders in a file system and attributes with files.

XPath was developed to address elements and attributes in an XML document. Although XPath is a separate language, it is not used alone; it is always used in conjunction with XSLT or XPointer. The purpose of XPath is to address parts of an XML document. With XPath, you can select a single item or create a path expression that matches several items. This matching capability is very important to XSLT; it is the basis on which elements are selected by XSLT to be transformed. XPointer is the XML equivalent of hyperlinks in HTML. A link in XPointer is defined using XPath.

Note XPath is an essential part of XSLT. You will learn more details about it on Day 3, "Selecting Data."

The Benefits of XSLT

The benefits of using XSLT are closely related to the benefits of using XML. Assuming that data is either stored or communicated as XML (which is what XML is for), the benefits of XSLT are as follows:

- Retrieving data from data in an XML document
- Formatting data from an XML document for display
- Translating between an XML document used for communication and a format used within a system

XML and XSLT in Data Storage

When you need to store data, XML is very flexible. You can adjust it to easily fit the type of data it is supposed to store. In that regard, it is much handier than a relational database because a database is limited to related tables. Each table contains rows with a fixed number of columns, with each column having a fixed meaning. The result is that within a database, data is bound to a fixed format. The trouble is that not all data fits nicely in this fixed format. XML has a flexible hierarchical structure that can mimic many, if not all, existing data structures. For example, you can easily create an XML document containing an entire database.

When data is stored in an XML format, XSLT can be used to retrieve data from that document. The fixed format of a database makes it hard to query data from different tables in one operation. Because XML doesn't have this rigid structure, an XSLT document can easily gather data from different sections in an XML document.

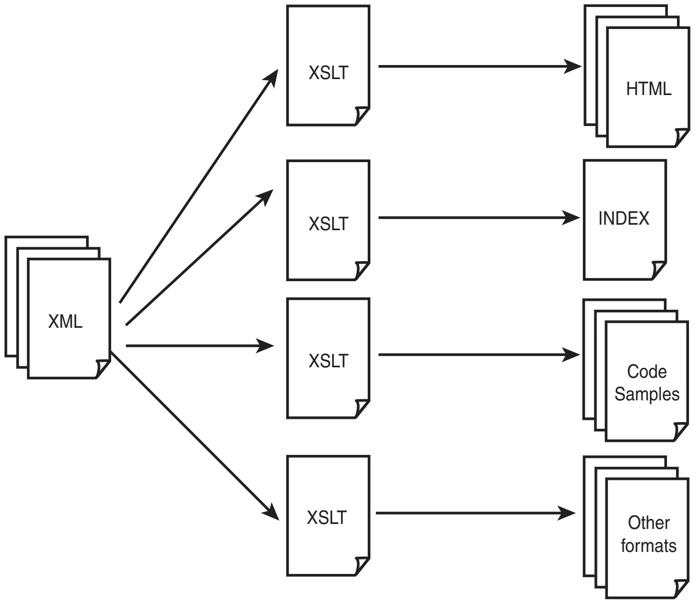
Because XML is so flexible, it can also store relatively unstructured data, such as text documents with some kind of formatting. As I mentioned at the beginning of this lesson, the benefit of detaching the information from the formatting is that you can search based on contextual meaning. With most programming languages, creating formatted output from such a tagged format is hard, whereas XSLT is actually designed to do this. Listing 1.2 shows a sample of such a tagged document that can be formatted with XSLT.

LISTING 1.2 Article Tagged in XML

```
<document xmlns:code="http://www.aspnl.com/xmlns/code" xml:lang="en-us">
  <title>Hello world sample</title>
  <text>
    This sample shows how to use <keyword>Response</keyword>.
    <keyword>Write</keyword> to write text to a <device>browser</device>
  </text>
  <code:block multiline="yes" type="lesson" subject="ASP"
    name="write" language="ASP">
    <code:html>
      &lt;html&gt;
      <tab />&lt;body&gt;
    </code:html>
    <code:asp>
      <code:keyword>Option</code:keyword>
      <code:keyword>Explicit</code:keyword><br />
      <br />
      <code:comment>'declare variable(s)</code:comment>
      <code:keyword>Dim</code:keyword> strWrite<br />
      strWrite = "Hello World!"<br />
      <code:object>Response</code:object>.
      <code:method>Write</code:method> strWrite
    </code:asp>
    <code:html>
      <tab />&lt;/body&gt;
      &lt;/html&gt;
    </code:html>
  </code:block>
</document>
```

ANALYSIS The XML in Listing 1.2 is part of a document used to create HTML for a Web site with color-coded code samples. You may be wondering why I didn't create the HTML directly instead of creating it from XML because creating the HTML from XML requires an extra step. The answer is that XML is not only used to create HTML files, but also the index file and code samples that people can run to see the result. Also, the same XML can be used to create a printed manual that could be used in a class. The separate files are created using XSLT. Each output type is created using a different XSLT document, and each XSLT document is a template for its output type. Any document that uses the same XML tags can be transformed into that output type using the same XSLT document, as depicted in Figure 1.1. Although in the beginning you need to do some extra work creating the XML and XSLT, in the end you will save a lot of time because you can reuse the XSLT to create all the files you need. This makes XML and XSLT very useful in document management and Web site management scenarios. The latter case is especially true for Web sites that target multiple platforms, such as handheld devices, in addition to regular browsers.

FIGURE 1.1
XML transformation to multiple outputs.



Besides targetting multiple platforms using one XML document and several XSLT documents, you can also create a unified look and feel for multiple XML documents. That means you have to create only one XSLT document that covers the look and feel of an entire site. An additional benefit here is that you can do this for multiple languages. So, creating a Web site that can be viewed in multiple languages is much easier than with other approaches. After all, the data storage is the same and so is the XSLT document.

An additional benefit of storing data in an XML document is that it can be queried using XPath. This procedure works somewhat like selecting data from a database using Structured Query Language (SQL). The XML document therefore is used somewhat like a database itself.

A problem with this usage occurs when you start working with multiple users. If a person editing a document locks it, someone else cannot read from and query it. This makes XML unsuitable for use in a multiuser environment. Databases (and database servers in particular) are designed for use in a multiuser environment but can't handle XML very well.

You can dump an XML document into a database as a text field, but then you would need to extract it from the database before you could query it. For this reason, XML support is making its way into database technology more and more. Some databases enable you to extract the relational data stored in it as XML. Although this capability is a start, it is not as good as a database that allows native support for XML storage and makes the entire XML queryable with XPath. These types of databases are in existence, though, and slowly getting better. They do not yet come close to the speed of relational database systems, but they are improving fast. With these databases, many existing types of applications could be created more easily, and possibly you can create new applications that are as yet beyond your reach. You can find more details about XML databases at `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`

XML and XSLT in Communication

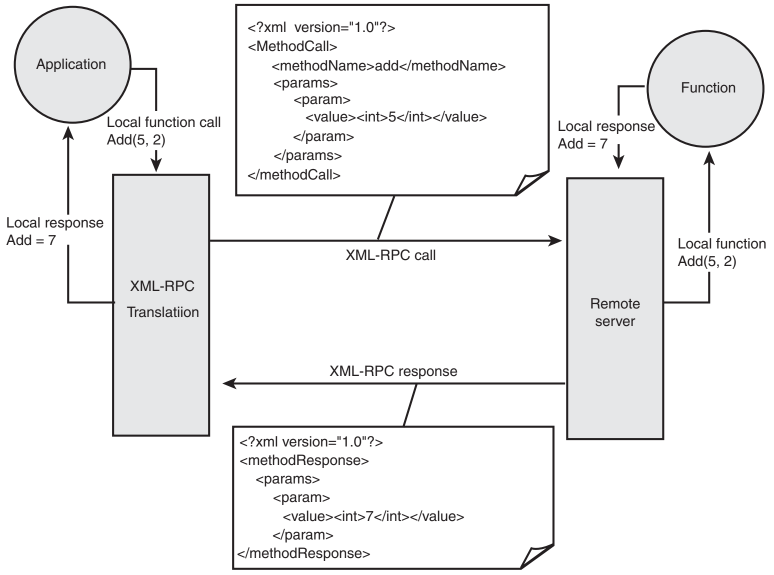
New Term Because XML is a nonproprietary data format that is based on one of the most basic data types, the string, it can be read by almost any computer. This makes it the ultimate data format for communication between systems. Any system equipped with an XML parser can consume and use XML. Web Services are based on this concept. A *Web Service* is a function provided by one system and usable by another system across the Internet.

A Web Service differs from a Web page in that a Web page is meant for display, and the result of a Web Service in most cases is not direct. The results of a Web Service may not see the light of day for a long time after it has been used, but it is equally possible that the result is used to display a composite result right away.

Web Services can be implemented with a number of technologies. Two of the most-used technologies are XML Remote Procedure Calling (XML-RPC) and the Simple Object Access Protocol (SOAP). Both define a protocol (or message format) allowing a system to use a function on another system, as shown in Figure 1.2. Although the two technologies are the same in nature, they use two different XML formats. As long as an application using the service knows which protocol it is dealing with, it can retrieve data from that format. If the application needs to manipulate the data, it either must rely on the XML DOM or use XSLT.

FIGURE 1.2

An application making a function call across the Internet.



```
<?xml  version="1.0"?>
<MethodCall>
    <methodName>add</methodName>
    <params>
        <param>
            <value><int>5</int></value>
        </param>
    </params>
</methodCall>
```

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><int>7</int></value>
      </param>
</methodResponse>
```

The type of communication that XML-RPC and SOAP implement isn't new. A few existing technologies have the same function. These technologies, including CORBA/IIOP, Microsoft COM/DCOM, and Java Remote Method Invocation, are all pretty much system specific, however. These technologies also do not use the Hypertext Transfer Protocol (HTTP) for communication between systems and therefore do not pass through a regular firewall protecting networks. These methods ,work only if the firewall is specifically configured to allow other protocols. XML-based communication systems are not system specific, because any system can deal with string data. With XML-RPC and SOAP, the communication can take place through HTTP, so it will work fine, even if a firewall is in place. XML-RPC and SOAP are not restricted to HTTP; however, e-mail or some other means of communicating the messages are also possible.

BizTalk Framework

BizTalk is an architecture that takes the concept of communication between systems with XML even further. It is an architecture for Enterprise Application Integration, based on XML. It is designed to interact with several systems in a business process. Each system involved in the process may use different XML formats, each of which is interpreted by a server that orchestrates the process. If necessary, this server can transform these messages to another format so that they can be understood by another system. An implementation of the BizTalk architecture would hardly be possible without XML to interact between the systems and XSLT to transform messages from one form to another. If you want to know more about BizTalk, check out `http://www.biztalk.org`.

Some older protocols, such as Electronic Data Interchange (EDI), are adopting XML for communication as well. Chat applications that use XML are already available. Someday you may see most communication protocols replaced by a single, XML-based protocol.

XML can also be used to communicate within an application. A major benefit to this approach is that every component in the system can access data passed within the system in the same manner, no matter what kind of component it is. Data from different kinds of data sources also can be represented in the same way. Therefore, no matter if the data comes from a database, mail server, or another source, the data representation is the same. XSLT can be used to iron out the differences and transform the data into a format suited for all the sources. The benefit is that applications are not confronted with interface differences of the underlying system. This also holds true for functions interacting through XML.

When Not to Use XSLT

Although XSLT is a powerful tool in modern data-driven applications, it is by no means the answer to *all* your problems. In some situations, other approaches make much more sense. Providing a complete list of situations is not possible, but I will discuss some of the more common applications in the following sections. This information will give you an idea of the situations you should avoid, or at least think twice about. You will need to use this information as well as your experience to judge whether XSLT is the right solution for a problem you're trying to tackle.

XSLT Performance Problems

XSLT is extremely useful in applications in which data conversion is key, such as document management and publishing applications. Because of the performance, XSLT is less useful in applications that require a lot of processing. The performance aspects of XML and XSLT will change, however, as applications become more and more centered around them. Applications that don't suffer much from these performance problems have a distributed nature, in which the transformation can be performed on the client. With the key browser manufacturers including XML and XSLT support in their browsers, Web applications that use XML and XSLT can utilize the power of the client and possibly reduce network traffic. This approach is far more appealing than a server having to do all the transformation to a format that can be read by the client, as the transformation process in such an application could become a major bottleneck.

A specific scenario that suffers from performance problems is Web sites using XML and XSLT. As yet, the major browsers do not properly support XSLT. Performing transformations in the browser, distributing the load, is therefore not possible yet. Transforming all the XML at runtime in a busy Web site is far too slow to be a viable solution. Preliminary benchmarks with the XSLT processor in Microsoft's .NET Framework suggest, however, that this situation may change sooner rather than later.

Data Warehousing Applications

Currently, both performance and concurrency issues make XML and XSLT less suitable for data warehousing applications. Although native XML databases exist, they can't compete with the top relational databases that have been around for a while. If you can structure your data to be stored in a relational database, however, there is the possibility of extracting the data as XML if you use some of the current relational databases. SQL Server 7 and Oracle provide add-ons that enable you to return a query result as XML. SQL Server 2000 provides this capability by default and can be configured to do so over the Web.

Computational Applications

XSLT also fails to deliver the goods in highly computational applications. Although XSLT certainly provides computational capability, most programming languages, such as C and FORTRAN to name a few, offer far more functions for complex computations and with much better performance.

Using CSS Instead of XSLT

In scenarios in which your XML documents are targeted at the Web, XSLT might be overkill. If the data doesn't need to be filtered and is in the right order, Cascading Stylesheets (CSS) do the job very nicely. CSS is by no means restricted to use with HTML. In fact, you can "invent" tags in HTML and attach a style to them. Because for all intents and purposes XML tags can be seen as HTML tags that you invented yourself, attaching a style is as easy as it is in HTML. In this scenario, using XSLT is actually counterproductive. Using CSS is a quick solution that requires no knowledge of XSLT and can probably be handled by most Web designers.

How Does XSLT Work?

With a general view of XSLT under your belt, it's time to move on to the actual workings of the language. Before you actually start to work with XSLT, however, you must understand some of the basics about how it works.

NEW TERM The transformation process of an XML document is performed by a *processor*, which is an application (or software component) that reads an XML document and an XSLT document and applies the XSLT to the XML. Processors exist both as command-line–runnable applications and as software components that can be used in an application. In the next section, I will discuss some of the more common processors and how they are used.

A processor consumes XML and, as such, is built on an XML parser. This parser can load the XML and XSLT documents using DOM and then apply the XSLT to the XML. Another option is a processor based on SAX.

Note My discussion of XSLT transformation at this point is purely theoretical and loosely based on the DOM approach. Actual processors most likely do not follow this exact process.

XSLT Transformation Explained

To understand the actual transformation process, examine a sample transformation. Listings 1.3 and 1.4 show a sample XML document and the XSLT document to be applied to it.

LISTING 1.3 Sample XML Document with Pets

```
<?xml version="1.0" encoding="UTF-8" ?>
<pets>
  <pet type="cat">Max</pet>
  <pet type="parrot" color="red">Peter</pet>
</pets>
```

ANALYSIS Listing 1.3 is a simple XML document representing my pets (actually, I don't have a parrot). The pet names appear between the `<pet>` tags, which have a `type` attribute denoting the pet type (in this case, a cat and parrot). For the parrot, I also defined a `color` attribute. Figure 1.3 shows a tree representation of Listing 1.3.

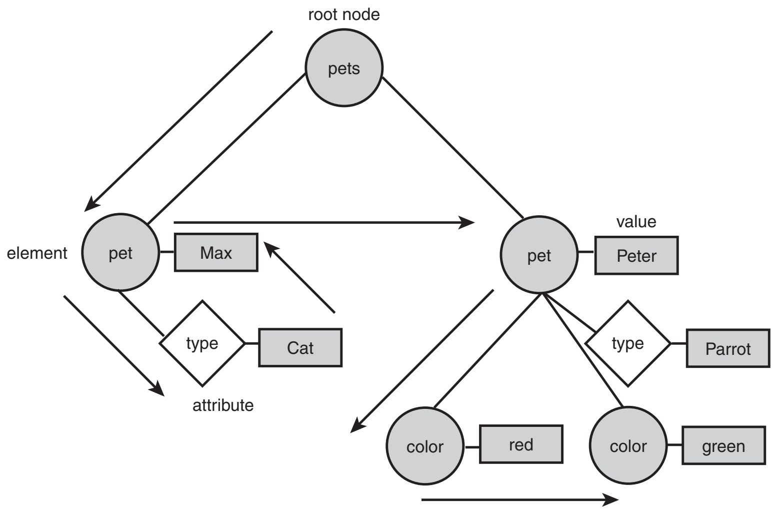


FIGURE 1.3
Node tree representing Listing 1.3.

New Term In Figure 1.3, the circles represent elements (tags) in the XML source. The diamond shapes represent attributes (of a tag). The rectangles represent values of either the element or attribute they are associated with. Within this tree, the circles and diamonds are known as *nodes*. When this tree is transformed using XSLT, the processor starts with the root node and "walks the tree" in the direction shown by the arrows. When the processor encounters a node, it searches for a rule in the XSLT document, matching the name and location within the tree of that particular node. If it finds such a rule, that rule is then applied to that node. This means that the execution of XSLT doesn't have a step-by-step sequence that is common to languages such as C, FORTRAN, and COBOL. These languages are procedural in nature, following a predetermined sequence of commands. Object-oriented languages, such as C++, Java, and Visual Basic, are based on the same model, except that the sequences are part of operations on objects. The one extra feature that object-oriented languages add to this is *event-driven* execution, which means that code is executed when some event happens—for instance, when you click a button.

New Term The execution of XSLT is similar to event-driven execution. In this type of execution, an event determines the sequence in which code is executed. In XSLT, this sequence is determined by the data that is encountered. This is why XSLT is based on *data-driven* execution, which means that code is executed when a certain piece of data is encountered.

Listing 1.4 contains a simple XSLT document that you can use to transform the XML in Listing 1.3. It consists of four rules that determine whether the code should be executed.

Listing 1.4 Sample XSLT Document for Pets

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" encoding="UTF-8"
    omit-xml-declaration="yes" />

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

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

  <xsl:template match="pet">
    My <xsl:value-of select="@type" /> is called <xsl:value-of select="." />.
    <xsl:apply-templates select="@color" />
  </xsl:template>
  <xsl:template match="@color">
    <xsl:value-of select=".." /> is <xsl:value-of select="." />.
  </xsl:template>

</xsl:stylesheet>
```

The output for Listing 1.4 looks like Listing 1.5.

OUTPUT **LISTING 1.5** Output When Applying Listing 1.4 to Listing 1.3

```
My cat is called Max.


My parrot is called Peter.
Peter is red.
```

Note Unless otherwise stated, all output is the result of using (Instant) Saxon version 6.2.2. This program and several others will be discussed later in this lesson.

Note The whitespace in between the lines of text in Listing 1.5 is supposed to be there. It appears based on the way XSLT handles whitespace by default. On Day 7, "Controlling the Output," you will learn how to remove the whitespace.

ANALYSIS The first rule `<xsl:template match="/">` is applied when the processing starts, as it matches the root of the document. Within this rule, the command `<xsl:apply-templates />` tells the processor to go on to any child node or nodes. In this sample, the child node is `pets`, which tells the processor to do the same again. This brings the processor up to the first `pet` node, firing the appropriate template, the first of which actually generates output. The second child node is processed by the same rule, which goes on to fire the `color` attribute rule.

As you can see, the output is different for the first `pet` and the second because the first doesn't have a `color` attribute. This is what XSLT is all about. If there is no node (element or attribute), nothing happens. But if there is, the rule is applied. If you have programming experience, you might not think this is much different from a program that reads input and acts on it. Such a program, however, contains explicit commands explaining what to do with certain input. XSLT works the other way around: The commands are there, but they're used only when applicable.

Programming event-driven code can be tricky because you don't know in which order events may occur; thus, you don't know in which sequence your code will be executed. On the other hand, the event-driven model more closely resembles what happens in most applications. Writing data-driven code can be even harder because code is executed in a nondeterministic way, depending on whether certain data exists and the data's location within the document tree. If the location of some piece of data is different, so is the order of execution. The result, however, need not necessarily be different.

Understanding Declarative Programming

NEW TERM So far, you've seen that XSLT is a rule-based programming language. Another major difference between XSLT and some of the more common programming languages is that XSLT also is a declarative programming language. With a *declarative programming language,* you tell the computer what to do rather than how to do something.

Declarative programming languages abstract the steps the computer has to take from what you want the computer to do. Instead of programming the steps the computer has to take, you specify what you want to happen. Arguably the most-used declarative language is Structured Query Language (SQL). SQL specifies which data you want to get from a database. It doesn't tell the database how to get the data. That job is left up to the query processor. In a similar fashion, XSLT leaves the "how" up to the processor. This actually accounts for the different parser/processor models. The difference between the "how" with DOM- and SAX-based processors is huge, yet XSLT works equally well on both types without any modification. Similarly, database servers can implement the storage of data completely different from one another. As long as they can understand SQL, the result is the same.

Like rule-based programming, declarative programming takes some getting used to. With nondeclarative languages, you have tight control over what happens because you specify every step of the process. Because you do not specify every step in a declarative language, you have less control over the result (or that's what you will think anyway). The advantage of declarative programming is that, in general, the programming process will go faster. When you want a slightly different result from the general case, programming will take more effort because you will have to work with a different sort of toolset. Whereas in traditional languages you could just change a processing step, here you are stuck with what the processor comes up with. You have to find a way to use the tools the language provides.

You can compare declarative programming with coaching a football team. As the coach, you don't compete in an actual game. You just tell the team members what you want them to do. How they do it depends on what happens in the game. The difference between a computer and a football team is that the computer will do exactly what you tell it to do, whereas a team may not. However, even if the computer does exactly what you tell it to do, the result may not be what you expected. Remember Listings 1.3 and 1.4. There, I just wanted to get two lines of text; instead, I got more lines and some whitespace I didn't expect. To get the whitespace out of the way, I obviously have to specify more closely what I want.

Creating XSLT Files

You have to create XML and XSLT documents before you can do anything. To create both, you have many options; the most common are discussed in the following sections. Most are Windows applications, but some are available for other platforms or are Java based and will run on any platform running the Java Runtime Environment.

Using a Text Editor

You can create XML with a number of tools. Because it is just text, you can use any text editor, such as Notepad, Textpad, or UltraEdit. The advantage of using a text editor is clearly that you can quickly make changes because these programs are lightweight and load fast. Also, text is easily editable. However, using a text editor is not the best approach to writing XSLT (or XML). A major disadvantage is that XML and XSLT have to be structured properly, and the commands need to be accurate. When you use a text editor you can easily make typing errors, and forgetting a closing tag is also common.

Using an XML Editor

XML editors come in different forms. The most basic ones offer color-coded display and possibly syntax checking. These editors are often part of a development environment used to create Web sites. A good example of such an editor is Allaire Homesite. Starting with version 4.5.2, it has standard XML tag support, and for earlier versions, extensions are available.

Other types of editors offer an interface in which you can easily edit nodes within a document. These editors represent the XML document as a tree view that you can manipulate. The most basic editor of this type is XML Notepad, which you can freely download from `http://msdn.microsoft.com/xml/`. XML Pro version 2.01 from Vervet Logic, (`http://www.vervet.com`) also supports this type of interface and has DTD support.

The third type of editor combines source code editing and node tree editing. This type of editor gives you the advantage of less error-prone tree manipulation but at the same time offers you more control over the actual source code. When you get into more specific output, this control is very important. At the time of this writing, one of the best editors is XML Spy (`http://www.xmlspy.com`), which offers many options. With the useful auto-complete option, you can easily pick XSLT elements and attributes. Because auto-complete is context-sensitive, it shows you only the elements or attributes that are applicable at your location in a document. XML Spy is pluggable, so you can configure it with any parser you want.

The problem with most regular XML editors is that they have no embedded XSLT support. This means that they cannot validate your XSLT tags by default. If they support validation against a Schema or DTD, you can implement this validation yourself. Validating your XSLT documents is paramount if you want to be able to quickly create XSLT documents. Validation cuts down on mistakes such as wrong XSLT tags or attributes that aren't supported by a specific tag.

XSLT Editors and Debuggers

Some XSLT editors and debuggers are currently available. These applications offer much more than an interface that allows quick creation of XSLT documents. They offer debugging options; for example, they allow you to perform a transformation step by step, with each step showing you which rule is fired. The major advantage of this capability is that you can see what is happening and possibly where you're going wrong.

eXcelon Stylus Studio

eXcelon Stylus Studio (`http://www.stylusstudio.com`) enables you to write XSLT completely by hand, aided by an auto-complete tool that shows the available XSLT elements and attributes. Another option is to write the XSLT document only partially by hand, based on an existing XML file. From the tree representation of the XML file, you can also create rules that should be applied to that element. With the built-in parser, you can then step through the transformation process. Other processors can be plugged in, but stepping is not supported in that case.

Marrowsoft Xselerator

Marrowsoft Xselerator (`http://www.marrowsoft.com`) is not quite as feature rich as Stylus Studio, but it does offer a context-sensitive auto-complete. It offers only those XSLT elements that should be available in the part of the document you're working in. Like Stylus Studio, it provides stepping and pluggable processors. Xselerator is very easy to use and gives you quick results.

XSL Debugger

An intriguing alternative to the commercial products described in the preceding sections is XSL Debugger, developed by TopXML.com (`http://www.topxml.com`), a community Web site on XML. Although limited as an editor, it provides solid debugging, which makes it useful as an extra development tool if you're already using an editor such as XML Spy.

Visual Studio.NET

Visual Studio.NET itself does not provide any XSLT debugging capabilities. However, Visual Studio.NET is highly pluggable, giving third-party vendors the opportunity to create something. Active State (`http://www.activestate.com`) has developed Visual XSLT as a plug-in to Visual Studio.NET. This plug-in provides XSL Debugger–type debugging and tag validation.

XSLT Design Tools

The last type of editor available goes at XSLT creation from the opposite direction. By using an existing XML document, you can create XSLT documents in a WYSIWYG environment. Using this editor is more or less like working with a WYSIWYG HTML editor, but with some added functions to work with XML documents. Whitehall <xsl> Composer (`http://www.whitehall.com`) is the only such product available on the market. <xsl> Composer is impressive because you don't need any knowledge of XSLT to use it. However, as with all such environments, getting exactly what you want is very hard.

Processors for XML Transformation with XSLT

Many processors are currently available, and still more are under development. There is no need to discuss them all, so the discussion here is limited to some of the more popular processors: MSXML, Saxon, and Xalan. Although these processors are free, you should read the license agreement before using them. You can find a list of other available parsers at `http://www.w3.org/Style/XSL/`. Another processor that is bound to be popular is the .NET Framework XSLT processor. It's not discussed here because it cannot be invoked from the command-line yet. I mention it, however, because preliminary tests show that this processor outperforms all the existing processors by a considerable margin.

MSXML

MSXML is the XML parser/processor available from Microsoft. The first version was shipped along with Internet Explorer 5.0. Because it was shipped before the XSLT specification was final, this version is not fully compliant. Versions 2.0 and 2.6 are a step in the right direction but are still lacking in a few areas. MSXML 3.0 and higher are good and used often by Visual Basic and Active Server Pages (ASP) developers. The latest version is available from `http://msdn.microsoft.com/xml/`.

MSXML is a component, so it cannot be run as a separate application. If you want to use it, you have to write an application. Having to go to all this trouble sounds pretty bad, but in 9 times out of 10, XML and XSLT will be used in a custom application anyway. However, Microsoft has provided a command-line executable called MSXSL, which you also can download from `http://msdn.microsoft.com/xml`.

Installing MSXML and MSXSL

MSXML comes in a Windows installation package. To install it, run the package and follow the installation steps. Because it has no options, your installation can't go wrong.

Note

MSXML does not come with any documentation. The documentation is part of the MSXML Software Developers Kit (SDK), which you can download from `http://msdn.microsoft.com/xml`.

MSXSL comes in a ZIP file. Apart from unpacking it, you don't have to perform an installation. You can run it from the command-line, but be sure that it is either in the same directory or that a path is defined to the directory holding the executable. The easiest way to ensure this is to place MSXSL.exe in the System or System32 directory of Windows. The ZIP file also contains a Word document discussing all the command-line options.

Running MSXSL

To run MSXSL, follow these steps:

1. Open the MS-DOS command prompt.
2. Change to the directory containing your XML and XSL files. You also can specify the full path to the files when you invoke MSXSL.
3. At the command prompt, type the following:

   ```
   msxsl source.xml stylesheet.xsl
   ```

   If the syntax of the documents is correct, the output is displayed.

As you can see, transformation from the command prompt is fairly easy. MSXSL command-line options will be discussed in Appendix C, "Command-Line Options for Common Parsers."

Saxon

Saxon is a Java-based XSLT processor developed by Michael Kay. It comes with a SAX parser but will work with other SAX parsers as well. Because it runs on Java, it will work on any system that has the Java Runtime Environment installed. For Windows users, an executable that can be run from the command prompt also is available. For programmers, Saxon offers an API that can be used with Java.

Installing Saxon

You can download Saxon from `http://users.iclway.co.uk/mhkay/saxon/`. There, you can choose from two versions: the full version and Instant Saxon. You need to run the full version under Java; you can run Instant Saxon from the Windows command prompt. Both require that the Java Runtime Environment version 1.1 or higher is installed.

After you unpack Instant Saxon, place it in a directory you want. Then run Instant Saxon from the command-line, either in the same directory as the executable, or from another directory if a path to the executable has been defined.

If you install the full version, make sure that the Java `classpath` environment variable has a reference to saxon.jar.

Running Saxon

To run Saxon, follow these steps:

1. Depending on your operating system, open the command prompt, a command window, or the shell.
2. If you're using Instant Saxon, type

   ```
   saxon source.xml stylesheet.xsl
   ```

   If you're using Saxon with Java, type

   ```
   java com.icl.saxon.StyleSheet source.xml stylesheet.xsl
   ```

   Providing the input is correct, the output should be displayed.

Xalan

Xalan is a processor developed by the Apache XML Project (`http://xml.apache.org`). The first version, Xalan-C++, is no longer available and has been replaced by Xalan-Java. You can download it from `http://xml.apache.org/xalan-j/index.html`.

Xalan runs on top of the Xerces-Java parser. It is pluggable, so it will run with other parsers as well. Like Saxon, Xalan offers an API so that you can use Xalan within Java applications.

Installing Xalan-Java

The Xalan-Java parser comes in a ZIP or GNU-ZIP package, which can be extracted to a directory you want. You then need to add a reference in the Java `classpath` environment variable that points to `xalan.jar` in the extracted package's bin directory.

Running Xalan-Java

To run Xalan-Java, follow these steps:

1. Depending on your operating system, open the command prompt, a command window, or the shell.
2. From the command prompt, run Xalan by typing

```
java org.apache.xalan.xslt.Process -in source.xml -xsl stylesheet.xsl
```

Summary

Today you learned that XSLT is a language used for manipulating and transforming XML documents. XSLT, which is itself XML, offers a vocabulary of commands that performs certain functions on an XML document. XSLT was developed as part of XSL but can be used separately. XSLT incorporates XPath to select and filter elements and attributes within an XML document.

XML and XSLT have their roots in SGML and DSSSL but are much simpler. Other technologies have been added to the XML family, so it is still growing. Some of these technologies, such as XML namespaces, have great bearing on XSLT.

XSLT is useful in document management scenarios in which you need multiple outputs of the same document. These target documents can be a range of types, such as plain text, HTML, XML, and PDF. When the Web is the only target, CSS may be a viable alternative. Because of concurrency and performance issues, XML is less useful for high-end data storage and for scenarios in which server-side transformation is required.

Many processors are available for XSLT, based on different parsers and different types of parsers (that is, DOM and SAX). The Java-based parsers can be used in Java applications and from the command-line. Microsoft also offers a parser/processor and an add-on to run it from the command-line.

Tomorrow you will learn the basics of XSLT and start working on your first transformation.

Q&A

Q Will XSLT replace CSS?

A Probably not. XSLT is more complex than CSS, so for simple documents, CSS is a good solution that more people know how to use. CSS also can create some effects that XSLT can't. XSLT and CSS can be used together to create richly formatted documents. A problem with CSS is that it operates only on element data. Data stored in attributes can't be displayed with CSS; with XSLT, it can.

Q It seems XML/XSLT has some drawbacks that must be overcome. Why should I start learning XSLT now?

A XML/XSLT is one of the fastest growing fields of technology at the moment, with most major corporations backing it. Many of the problems are known and are being addressed. They will be taken care of sooner rather than later. A good example is the XSLT debuggers. Until fairly recently, no XSLT debuggers were available, and not many people had an idea how to create one because XSLT works differently than languages like Visual Basic. Now several very good products are available.

When you start working with XSLT, you also will find that it can easily solve problems that now take you a lot of effort. XSLT, in that sense, is just another tool in your toolbox.

Areas in which XSLT will be of much benefit are applications targeting multiple platforms, applications in multiple languages, translation between different data storage formats, and applications working with relatively unstructured data that needs to be queried.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: XSLT can transform XML to XML, HTML, and different text-based file formats.
2. True or False: You have to use `xsl` as a namespace for XSLT.
3. What do you need to run XSLT?
4. XSLT is based on a data-driven programming model. What is meant by this?
5. XSLT is a declarative programming language. What is the difference between declarative languages and languages such as C, Java, and Visual Basic?
6. What kinds of tools can you use to create XSLT documents?

Exercise

1. Create an XML document with the following code:

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

```
  <pet type="cat">Max</pet>
  <pet type="parrot" color="red">Peter</pet>
</pets>
```

Now create an XSLT document with following code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" encoding="UTF-8"
    omit-xml-declaration="yes" />

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

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

  <xsl:template match="pet">
    My <xsl:value-of select="@type" /> is called
 <xsl:value-of select="." />.
<xsl:apply-templates select="@color" />
  </xsl:template>

  <xsl:template match="@color">
    <xsl:value-of select=".." /> is <xsl:value-of select="." />.
  </xsl:template>

</xsl:stylesheet>
```

Execute the files using any of the processors discussed today. If you use two different processors, will the output be the same?

DAY 2

Transforming Your First XML

Yesterday you learned what XSLT is and where it fits in between other technologies. You also learned how XSLT transforms XML and how to run XSLT with some of the available processors.

Today you will learn how to create your own XSLT document and apply it to an XML source. You will learn about the basic elements in XSLT and how they are structured in an XSLT document.

In today's lesson, you will learn the following:

- What a stylesheet is
- The basic structure of a stylesheet
- The basic elements of a stylesheet
- Different methods for applying a stylesheet to an XML source

Anatomy of a Stylesheet

An XSLT document has a certain structure and consists of some basic elements. A proper XSLT document needs to conform to this structure, and it must be built with the basic the elements.

What Is a Stylesheet?

As you learned yesterday, XSLT is short for Extensible Stylesheet Language Transformations. Until now, I have been talking about XSLT documents, but these documents are actually called *stylesheets*. This term is somewhat misleading, especially if you've been working with Cascading Stylesheets (CSS). In the context of CSS, a stylesheet defines the style or layout of certain HTML or XML tags. Using such a stylesheet, you can define the font type, size, and color of text, and also the border style, background color, and so on of tables and other nontext tags in HTML.

New Term A stylesheet in the context of XSLT does something different: It creates output from an XML document, which might (or might not) contain formatting information. The output also can contain only a subset of the data in the XML document. In addition, the data may be rearranged and restructured. These features go way beyond what you can do with CSS, and most of them have nothing to do with defining presentation formatting. This is all true because XSLT is part of XSL, which is all about presentation. XSLT is the part of XSL that deals with transformations. So, a stylesheet in XSLT is a document that transforms an XML document into another XML document. A stylesheet can also transform an XML document to HTML or text.

Basic Stylesheet Elements

Now that you know what a stylesheet is, you're ready to move on to the next step: understanding the basic building blocks of a stylesheet. Because XSLT is itself XML, these building blocks are themselves XML or related to it.

The XML Prolog

A stylesheet is itself an XML document, so if you really want to go by the book, your stylesheet should start with an XML document prolog, but this is not required. A typical prolog contains a processing instruction to tell a parser or processor the version and encoding type of the XML document. A typical XML processing instruction looks like this:

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

The XML processing instruction should always be on the first line of the document. The `version` attribute is mandatory, and its value should always be the current version of the XML W3Consortium (W3C) Recommendation. Currently, that version is 1.0. If the optional `encoding` attribute is used, its value should be the XSLT document's encoding type. Although this attribute is optional, I recommend that you always use it to ensure that special characters are handled properly. In regular cases, the processor can determine the encoding type correctly, but if it doesn't, your application could break or you might get some unexpected results. Valid encoding types are as follows:

- Unicode: UTF-8 or UTF-16
- ISO/IEC 10646: ISO-10646-UCS-2 or ISO-10646-UCS-4
- ISO 8859: ISO-8859-1, ... ISO-8859-*n* (*n* is the part number)
- JIS X-0208-1997: ISO-2022-JP, Shift_JIS, or EUC-JP

Note Optionally, a prolog can contain the `standalone` attribute. This attribute is rarely used and is beyond the scope of this book.

The Stylesheet Element

An XML document always has only one root element. In the case of a stylesheet, this element is appropriately named `xsl:stylesheet`. A typical `xsl:stylesheet` element looks like this:

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

As with the `version` attribute of the XML declaration discussed in the preceding section, the `version` attribute of the `xsl:stylesheet` element is required, and its value must be a valid version. At the time of this writing, the only valid version is 1.0; a new version is under development by the W3 Consortium, but it's still in its early stages of development.

The second attribute is not XSLT per se; it is a namespace declaration common to all XML documents using namespaces. Because XSLT uses a namespace to operate, you must always declare its namespace in the root element of the stylesheet.

Note Namespaces will be discussed on Day 15, "Working with Namespaces."

The value of the namespace declaration is very important. It is a URI that should point to a unique location. The URI used in the namespace declaration for XSLT is very specific and must be the same as in the preceding sample.

Caution

Do not remove, change, or misspell the namespace declaration. If it doesn't match the preceding declaration exactly, the stylesheet is not recognized as XSLT by the processor and therefore will not work.

Note

In the preceding code, the `xsl:stylesheet` element spans two lines. Because XSLT is XML, this is equally valid. Wrapping elements over more than one line can, in some cases, improve the readability.

The Internet Explorer 5 MSXML Processor

Internet Explorer 5 comes with the first MSXML version, which uses a different XSL syntax. This syntax, which is based on the December 1998 XSL working draft, uses a different URI: `http://www.w3.org/TR/WD-xsl`. These two versions of the syntax aren't compatible, so you must be sure to use the correct syntax and correct URI when you write stylesheets. If your application must run on Internet Explorer 5, be aware that many functions might not work or might not work correctly.

When you install a newer version of MSXML, it does not automatically replace the version used by Internet Explorer 5. This means that Internet Explorer will still use the older version, although a newer version has been installed. You can download the Xmlinst.exe tool from `http://www.microsoft.com/xml` to install the new parser in *replace mode*. You then can use Internet Explorer 5 with the new version. Be aware that you have to install the new version before you run the replace mode tool.

The `xsl:stylesheet` element can have more attributes, but at this point discussing them would be more confusing than helpful. These attributes will be discussed in Week 3.

Note

In this book, I will refer to a stylesheet and the `xsl:stylesheet` element as stylesheet because they are, in essence, the same. Where the distinction is relevant, I will use the term *`xsl:stylesheet` element*.

The `transform` Element

You may come across stylesheets that have another root element called `xsl:transform`. This element is exactly the same as the `xsl:stylesheet` element, and they can be used interchangeably. A stylesheet using the `xsl:transform` element has a root element that looks like this:

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:transform>
```

You may wonder why the same element has different names. There is no reason for having both elements, other than to keep all the committee members who developed XSLT happy.

Note

In this book, all the samples will use only the `xsl:stylesheet` element. I will make no further reference to the `xsl:transform` element apart from the XSLT element and function reference in Appendix B.

Stylesheet Contents

The `xsl:stylesheet` element as defined earlier is empty. Therefore, you would expect that it does nothing when a processor runs it. You would expect that any XML document that the stylesheet is applied to will yield an empty document. This, however, is not the case because of the default behavior of XSLT when no rules apply. If you process the XML document in Listing 2.1 with the empty `xsl:stylesheet` element, you end up with a result similar to Listing 2.2.

LISTING 2.1 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8" ?>
<pets>
  <pet type="cat">Max</pet>
<pet type="parrot">Peter
    <color>red</color>
  </pet>
</pets>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

OUTPUT **LISTING 2.2** Output from an Empty Stylesheet Applied to Listing 2.1

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

  Max
Peter
    red
```

Note

The output in Listing 2.2 was created with Saxon, which is the case throughout this book unless stated otherwise. The output depends on the processor you use. You might notice some differences in the whitespace generated before and after the text, as well as the output encoding. Saxon and Xalan, by default, generate XML in UTF-8 encoding, whereas MSXML generates XML in UTF-16 encoding. If you run Listing 2.1 from the command line with MSXSL, you get more or less the same result, but with a space between all the characters because UTF-16 is two-byte encoding, whereas UTF-8 is single-byte encoding.

ANALYSIS The result in Listing 2.2 contains an XML declaration on line 1, followed by the text contained in the elements of the source XML. The attributes' values do not appear in the result. Note that the spaces on lines 3 and 5, and linefeeds are in perfect sync with the spaces and linefeeds in the source XML. This is the result of the built-in rule that is used when no other rules apply.

Creating Your Own Rules

From what you have seen so far, the built-in rule doesn't appear to be very helpful. It actually is very helpful, but only if its behavior can be controlled. Creating your own rules is the first step in controlling the built-in rule and thus the output. You can define your own rules by using the `xsl:template` element, which is probably the single most important element in XSLT. In fact, it is so important that Day 4 is all about using templates. For now, the basics will suffice.

Note

In this book, I will refer to the `xsl:template` element as *template*. I will use the term *`xsl:template` element* only if it is relevant.

Using the `match` Attribute

A template works by matching the current element or attribute that the processor encounters in the source XML to the template's matching rule. If the template's rule matches the current element or attribute, that template is fired and applied to that element or attribute. The rule used to match elements and attributes is defined using a template's `match` attribute. The `match` attribute can contain a complex rule that matches many elements or attributes, but it can also contain a simple rule, matching only a single element. A template matching an element named `pets` in an XML document looks like this:

```
<xsl:template match="pets">My pets</xsl:template>
```

If this rule is applied to an XML document containing a `pets` element, for example, each time this element is encountered, the text `My pets` is written into the result. Listing 2.3 shows a complete stylesheet with this template rule.

LISTING 2.3 Stylesheet Matching Only `pets` Elements

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="pets">My pets</xsl:template>
6:
7:  </xsl:stylesheet>
```

Note

The indentation and linefeeds in Listing 2.3 have been added for readability. This convention is used by most developers and will be used throughout this book.

ANALYSIS Listing 2.3 starts with a prolog on line 1, followed by the mandatory `xsl:stylesheet` element as the root element of the stylesheet. Line 5 contains a template that matches any `pets` element in the source document to which the template is applied. If you apply Listing 2.3 to Listing 2.1, the output is similar to Listing 2.4.

OUTPUT **LISTING 2.4** Result from Listing 2.3 Applied to Listing 2.1

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

ANALYSIS Although Listing 2.4 doesn't provide a startling result, it is significant. First, you can see that the rule in Listing 2.3 matching any `pets` element is fired once; this corresponds to Listing 2.1, to which you applied Listing 2.3. Note also that the built-in template rule is applied until the template matching the `pets` element is encountered. That template is executed, so the text `My pets` is inserted. Other than that, nothing happens. If no template is matched, the built-in template rule makes sure that the child elements are processed next. When a template does match, it must tell the processor to continue processing the child elements explicitly, which it doesn't in this case. The built-in template rule comes into play again only if the processor is instructed to continue with the child elements.

Now try something else. The preceding example fired the rule created for the `pets` element, but because it was the root element, the exact workings of the built-in template rule aren't entirely clear yet. Now look at Listing 2.5, which is similar to Listing 2.3. Here, the rule works on any `pet` element instead of a `pets` element.

LISTING 2.5 Stylesheet Matching Only `pet` Elements

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="pet">This is my pet.</xsl:template>
6:
7:  </xsl:stylesheet>
```

ANALYSIS Listing 2.5 is similar to Listing 2.3. The difference lies in line 5, which contains a template that matches any `pet` element. This template inserts some different text as well. If you apply Listing 2.5 to Listing 2.1, the result is similar to Listing 2.6.

OUTPUT **LISTING 2.6** Result from Listing 2.5 Applied to Listing 2.1

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

  This is my pet.
This is my pet.
```

ANALYSIS The rule defined in Listing 2.5 is fired twice, once for each `pet` element in Listing 2.1. The whitespace from the source document also is copied. So, what happened? The processor first encounters the document root. Because there is no rule, the built-in template rule fires, saying "copy the text value and process the child elements." The root has no text value; its value consists of only the child element `pets`. Again, there is no matching rule, so the built-in rule fires. Because the `pets` element doesn't have a text value, there is no output, except for whitespace. The next element is a `pet` element, so the rule in Listing 2.5 fires as it should, resulting in the first `This is my pet.` line. After the rule fires, control reverts to the previous rule , in this case the built-in template rule, resulting in another matching `pet` element and a second line of text. Now an interesting question arises: Why isn't the next step writing the value of the `color` element? This value isn't written because the defined template doesn't instruct the processor to do anything with child elements. The built-in template rule doesn't apply because a defined rule is firing. The built-in rule searches only for immediate children of the current element for which it fires. It never reaches the `color` element because the defined template takes over control.

Now you can put the two rules together in one stylesheet to see what happens. If you fully understood the preceding analysis paragraph, you should be able to predict the output generated when the combined stylesheet is applied to Listing 2.1. Listing 2.7 contains the combined stylesheet.

Listing 2.7 Stylesheet with Rules for `pet` and `pets` Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="pets">My pets</xsl:template>

  <xsl:template match="pet">This is my pet.</xsl:template>

</xsl:stylesheet>
```

Analysis Applying Listing 2.7 to Listing 2.1 yields the same result as applying Listing 2.3 to Listing 2.1. (Is that what you predicted would happen?) You get the same result because the built-in rule encounters the `pets` element and fires the appropriate rule. Because all `pet` elements are child elements of the `pets` element, the built-in template rule never reaches them. The template defined for the `pets` element also doesn't contain a command that will tell the processor to invoke the built-in template rule for its child elements, so processing basically stops after the matching template deals with the `pets` element.

Matching the Root Element

A special element within XSLT is the document root of an XML document. Strictly speaking, the root isn't an element, but you can match it and act on it with a template. This point is important if you want to do something before you actually act on the root element of a document, especially when the root element can be different in different documents. A template matching the document root looks like this:

```
<xsl:template match="/">do something here</xsl:template>
```

You will see this element often in stylesheets because it allows you to generate output before XML source processing starts and after it is entirely processed.

Applying More Templates

If a template doesn't tell the processor to go further down into the hierarchy of the source document, processing the children of the current element, then no ancestor elements of the current element are ever matched, so they aren't processed. If you want the child elements of the current element to be processed, you should apply the built-in rule again for those elements so that they fire the appropriate templates. You can use the `xsl:apply-templates` element, as shown in Listing 2.8.

LISTING 2.8 Stylesheet Using `xsl:apply-templates`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="pets">
    My pets
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="pet">This is my pet.</xsl:template>

</xsl:stylesheet>
```

ANALYSIS In Listing 2.8, `xsl:apply-templates` is an empty element with no attributes. It simply tells the processor to apply the built-in template rule in the current context. Listing 2.9 shows the result when this stylesheet is applied to Listing 2.1.

OUTPUT **LISTING 2.9** Result from Listing 2.8 Applied to Listing 2.1

```
<?xml version="1.0" encoding="utf-8"?>
    My pets

  This is my pet.
This is my pet.
```

ANALYSIS As you might have expected, the output now contains the text of the `pet` template twice. It also contains the text of the template matching the `pets` element. Inserting the `xsl:apply-templates` element in that template ensures that the `pet` elements are processed and the appropriate templates fired, basically yielding a result like adding Listing 2.6 to Listing 2.4.

Note The whitespace in the result is still somewhat erratic. On Day 7, “Controlling the Output,” you will learn how to deal with this problem. For now, the whitespace helps you see what is actually happening.

Templates are fired only when they are matched to an element or attribute in the input. If a template does not match any element or attribute in the input, it is never fired. Listing 2.10 is nearly the same as Listing 2.8, but this listing has an added template.

LISTING 2.10 Stylesheet with a Template That Is Not Fired

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
➥Transform">
3:
4:    <xsl:template match="pets">
5:      My pets
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="pet">This is my pet.</xsl:template>
10:
11:   <xsl:template match="cage">My pet lives in a cage</xsl:template>
12:
13: </xsl:stylesheet>
```

ANALYSIS Listing 2.10 is similar to Listing 2.8, except that line 11 contains an additional template. That template matches any `cage` element. Applying Listing 2.10 to Listing 2.1 yields exactly the same result as applying Listing 2.8 to Listing 2.1; this output is shown in Listing 2.9. Because there is no `cage` element in Listing 2.1, the template on line 11 is never fired. Although this doesn't seem very significant, it is a very important concept in XSLT. This behavior enables you to create a stylesheet that contains many templates, matching different elements. Some templates will not be fired when you apply the stylesheet to an XML source because the elements matched by that template don't exist in that XML source. With another XML source, those templates may fire. So, different documents can be processed with the same stylesheet, generating different output, but with the same overall structure. In a Web site, you could use this feature to create HTML files with the same layout and formatting, but with different content.

Getting Data from an XML Source

So far, the resulting output from the stylesheets has contained only text from the stylesheet itself. The essence of XSLT, however, is that you can manipulate the data in an XML document. The most basic manipulation is getting the data values and writing them into the result. You can use the `xsl:value-of` element, which has a `select` attribute that should contain an expression telling the processor which value needs to be written to the output. Listing 2.11 shows the `xsl:value-of` element in action.

LISTING 2.11 Stylesheet Writing Data from Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
➥Transform">

```

LISTING 2.11 Continued

```
4:    <xsl:template match="pets">
5:      My pets:
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="pet">
10:     <xsl:value-of select="text()" />
11:   </xsl:template>
12:
13: </xsl:stylesheet>
```

ANALYSIS Instead of writing some literal text to the output, the template on line 9, which matches the `pet` element, writes the text value of the matched element to the output on line 10. The expression in the `select` attribute uses the `text()` function to select the text value of the current element. When you apply Listing 2.11 to Listing 2.1, the result is similar to Listing 2.12.

OUTPUT **LISTING 2.12** Result from Listing 2.11 Applied to Listing 2.1

```
<?xml version="1.0" encoding="utf-8"?>
    My pets:

  Max
Peter
```

ANALYSIS The text value of the `pet` elements in Listing 2.1 are the names of the pets. You can see in Listing 2.12 that the `xsl:value-of` element writes the name of the pets to the output. The template matching the `pet` element is matched twice, once for each pet, resulting in the different names sent to the output.

The `xsl:value-of` element seems very simple, and in essence it is. However, the `select` attribute works somewhat like a template's `match` attribute. The effect is that this attribute doesn't necessarily evaluate to a piece of text, which is actually why the `text()` function is used in Listing 2.11, telling the processor that it needs to write the *text* value of the current element. If the `select` attribute evaluates to the current element's value, it includes the child elements because they are, in a sense, part of the element's actual value. If that is the case, the output is a concatenation of all the string values of the current element and all its ancestors. The stylesheet in Listing 2.13 shows this effect.

LISTING 2.13 Stylesheet Writing All Ancestors of `pet`

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="pets">
6:      My pets:
7:      <xsl:apply-templates />
8:    </xsl:template>
9:
10:   <xsl:template match="pet">
11:     <xsl:value-of select="." />
12:   </xsl:template>
13:
14: </xsl:stylesheet>
```

ANALYSIS Listing 2.13 uses the same structure as Listing 2.11. The `xsl:value-of` element on line 11 now doesn't use the `text()` function to get the text value of the element being processed but gets the value of the element being processed with a period. The period indicates the value of the current element. When you apply Listing 2.13 to Listing 2.1, the result is similar to Listing 2.14.

OUTPUT **LISTING 2.14** Result from Listing 2.13 Applied to Listing 2.1

```
1: <?xml version="1.0" encoding="utf-8"?>
2:     My pets:
3:
4:   Max
5: Peter
6:     red
```

ANALYSIS Line 6 of Listing 2.14 shows the value of the `color` element in Listing 2.1. Because no template matches it, and no templates are invoked to process the child elements of the `pet` elements, the result must come from the `xsl:value-of` element. Because it doesn't select the `text()` value of the element being processed, but the value of the element, the text value of all ancestor elements are also added to the output.

Another situation in which the result is somewhat different from what you might expect occurs when the `select` attribute evaluates to multiple elements. In that case, the first element's text value is written into the result. The other elements are basically ignored. Because this happens with more complex data selections, I will not discuss that situation here. Tomorrow you will learn all about selecting data; I'll revisit this topic then.

Simplified Stylesheet Syntax

If you have experience only with HTML and not with programming, XSLT may seem daunting to you. When members of the W3 Consortium started working on XSLT, they realized that if people familiar with HTML couldn't make the step to XSLT, it was less likely to succeed. To make the step to XSLT easier, the W3C developed a syntax that more closely resembles HTML. Listing 2.15 shows a stylesheet with this syntax.

LISTING 2.15 Stylesheet with Simplified Syntax

```
1:  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xsl:version="1.0">
2:  <head>
3:    <title>Simplified stylesheet</title>
4:  </head>
5:  <body>
6:    <h1>My pets</h1>
7:    <xsl:value-of select="/pets/pet" />
8:  </body>
9:  </html>
```

ANALYSIS In Listing 2.15, the simplified syntax closely resembles HTML and differs from what you have seen so far. Except for line 7, all the elements you see are HTML elements. Also, this listing does not have an XML prolog. Although you can include the prolog, it could be confusing to HTML developers not familiar with XML. The simplified syntax is also XML, so it needs to be well formed like any other XML document. Tags common to HTML that don't have closing tags should be replaced with their XML (or XHTML) equivalents. `<br>` should be replaced by `<br />`, `<hr>` should be replaced by `<hr />`, and so on. In addition, empty attributes, such as the `<option>` attribute `checked`, should be replaced by their XML equivalents. In the case of the `checked` attribute, you use `checked="checked"`.

You can see on line 1 that the `html` element has taken over for `xsl:stylesheet` as the root node. That's why the `xsl` namespace is declared within the `html` element. Note that the mandatory XSLT version is also included, but it is defined as part of the `xsl` namespace because the `html` element is not defined within the `xsl` namespace and the `version` attribute relates to that namespace.

The simplified stylesheet syntax takes the place of the template matching the root (`match="/"`). A stylesheet using the simplified syntax therefore cannot contain any elements that can be only child elements of the `xsl:stylesheet` element. This includes other templates. In other words, the simplified syntax is a one-template stylesheet, which limits its capabilities. You can see this use best when you look at the result of applying Listing 2.15 to Listing 2.1, as shown in Listing 2.16.

OUTPUT **LISTING 2.16** Result from Listing 2.15 Applied to Listing 2.1

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

      <title>Simplified stylesheet</title>
   </head>
   <body>
      <h1>My pets</h1>Max
   </body>
</html>
```

ANALYSIS In Listing 2.16, notice that only one of the pet elements is actually written into the output. This happens because the select attribute actually contains an expression that matches two pet elements. The default behavior of the value-of element tells the processor that it must process only the first element in this case. So, to get both values, you would need to insert an additional value-of element and be able to make a selection that more specifically selects a certain element. The method for doing that is the central topic of tomorrow's lesson.

The simplified stylesheet syntax has limited capabilities. It is limited to only one scenario, but more important, generating complex rule-based output is hardly possible. You could, however, start by using the simplified syntax if it suits your needs. If you need to create more complex output in the future, you can easily change the simplified syntax to the regular syntax by surrounding the simplified stylesheet with the xsl:stylesheet element and a template matching the root element of a source document. The only other change would be to remove the namespace and version declaration from the html element.

Note

Because of its drawbacks, the simplified stylesheet syntax will not be discussed further in this book.

Applying a Stylesheet to an XML Source

On Day 1, I briefly discussed executing an XML source and stylesheet using several processors. Although this approach works fine, getting the output from the command line isn't useful in several scenarios. The following sections discuss some of the alternatives.

Linking a Stylesheet to an XML Source

You link a stylesheet to an XML source by adding a processing instruction to the XML source, telling the processor where to find the stylesheet. In scenarios in which you explicitly run the processor, this method is not very useful because most processors require you to tell it which XML document and which stylesheet to use. In some scenarios, however, you can't tell the processor explicitly which stylesheet to use. The most common situation occurs when an XML document is loaded in a browser that supports XML. The browser is pointed to the XML document but cannot determine whether to apply a stylesheet to it and, if so, which one. You can use a processing instruction in the XML document to tell the processor which stylesheet to apply to remedy this situation. A typical processing instruction looks like this:

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

Caution

In the W3C specification, the `type` attribute should have the value `text/xml` or `application/xml`. In Internet Explorer 5 (with the older parser), it needs to be `text/xsl`.

You need to include this processing instruction in the prolog, typically after the XML declaration discussed earlier. Listing 2.17 shows what Listing 2.1 would look like if you link the stylesheet in Listing 2.11 to it using a processing instruction.

LISTING 2.17 Listing 2.1 with a Linked Stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xml" href="02list11.xsl"?>
<pets>
  <pet type="cat">Max</pet>
  <pet type="parrot">Peter
    <color>red</color>
  </pet>
</pets>
```

ANALYSIS If you view Listing 2.17 in a browser supporting XML and XSLT, the stylesheet in the file `02list11.xsl` is applied to it, providing the file exists in the same directory as the file containing Listing 2.17. You can also run Listing 2.17 with Saxon, which provides a command-line option that tells the processor to use the linked stylesheet. If you save Listing 2.17 in a file named `02list17.xml` (as it is in the sample download), you run it using Saxon like this:

```
saxon -a 02list17.xml
```

The command-line option `-a` tells Saxon that it must apply the stylesheet that is stated in the processing instruction on the second line of Listing 2.17.

Note

At the time of this writing, only Internet Explorer 6.0 and higher support XML and XSLT by default. If you want to test this example with Internet Explorer 5, install MSXML version 3.0 or higher in replace mode. Other browsers such as Netscape 6.0 don't have XSLT support. A plug-in for Netscape is available from `http://www.inlogix.de/nsplugin.html`. The Mozilla TransforMiiX Project (`http://www.mozilla.org/projects/xslt/`) is another alternative.

You might be thinking right now that including a processing instruction undermines the whole idea of having multiple stylesheets doing different things with an XML source, and you would be right. However, it is important to realize that most processors just ignore this processing instruction, unless specifically instructed to use it, as is the case with a browser. Also, you can insert this processing instruction more than once, with attributes telling a processor which to use under what circumstances. These attributes, however, are optional. The attributes used in Listing 2.17—`type` and `href`—are mandatory.

Note

Processing instructions, strictly speaking, do not have attributes because they really aren't XML elements themselves. The W3C documentation refers to these attributes as *pseudo-attributes*. Because they have the same function as regular attributes, I'll keep referring to them as *attributes*.

To insert multiple stylesheets, you first add a `title` attribute to the processing instruction. This title distinguishes the separate stylesheets from one another and enables you to address them in an application. The title may contain spaces. Along with the `title` attribute, you might want to add `alternate="yes"`, indicating that this stylesheet is not used by default, but only in certain instances. Your stylesheet should have only one link with the `alternate` attribute set to `no`, or with the attribute omitted. All other links must have `alternate="yes"`. A series of different stylesheet links looks like this:

```
<?xml-stylesheet type="text/xml" href="huge.xsl" title="huge" alternate="yes"?>
<?xml-stylesheet type="text/xml" href="big.xsl" title="big" alternate="yes"?>
<?xml-stylesheet type="text/xml" href="default.xsl"?>
```

The preceding alternate stylesheets could, for example, be used for people with impaired vision. An application using these stylesheets might provide the users with a choice between the two, based on the given title.

Targeting Multiple Media

An XML document can contain multiple stylesheet processing instructions that target different media. When an application targeting a specific medium opens the XML document, it can apply the stylesheet defined for that particular medium. If you use a printed medium, the stylesheet looks like this:

```
<?xml-stylesheet type="text/xml" href="print.xsl" media="print"
 title="style for printed media" alternate="yes"?>
```

The `media` attribute in this case defines that the target is a printed medium. A stylesheet that has different stylesheets for printed media, handheld devices, and a default (in this case, a browser) contains the following processing instructions:

```
<?xml-stylesheet type="text/xml" href="printstyle.xsl" media="print"
 title="style for printed media" alternate="yes"?>
<?xml-stylesheet type="text/xml" href="handheldstyle.xsl" media="handheld"
 title="style for handheld devices" alternate="yes"?>
<?xml-stylesheet type="text/xml" href="default.xsl"?>
```

Currently, the `media` attribute can contain only one medium. The W3 Consortium reports, however, that parsers need to be able to parse comma-separated `media` attributes, so they might be used in the future.

The media supported at the time of this writing have been defined in the HTML 4.0 and XHTML specifications. They are listed in Table 2.1.

TABLE 2.1 Available Media Descriptors

Media Descriptor	*Description*
`screen`	Nonpaged computer screens
`tty`	Media using a fixed-pitch character grid, such as terminals
`tv`	Televisions and the like
`projection`	Projectors
`handheld`	Handheld devices, such as mobile phones and personal digital assistants (PDAs)
`print`	Printed materials or print preview mode
`braille`	Braille tactile feedback devices for people with impaired vision
`aural`	Speech synthesizers
`all`	All (other) devices

Embedding a Stylesheet in an XML Source

In HTML, you can link to a CSS much as you would link an XML document to an XSLT stylesheet. In HTML, you also can embed the stylesheet in the HTML document.

In a similar manner, an XSLT stylesheet can be embedded in an XML document. Although this method isn't used much and in general is considered to counter the use of XSLT, I'll briefly discuss it.

The first step is actually creating the stylesheet as part of the XML document. Typically, the stylesheet's root element, `xsl:stylesheet`, is a child element of the XML document's root element. Because the stylesheet is pointed to by name, this is not required, however. The next step is creating a processing instruction that links to a stylesheet, with a reference to the internal stylesheet. The internal reference looks like a reference to an internal anchor in an HTML document. Listing 2.18 shows how to embed Listing 2.11 in Listing 2.1.

LISTING 2.18 Stylesheet Embedded in an XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xml" href="#petstyle"?>
<!DOCTYPE pets [<!ATTLIST xsl:stylesheet id ID #IMPLIED>]>
<pets>
  <pet type="cat">Max</pet>
  <pet type="parrot">Peter
    <color>red</color>
  </pet>

  <xsl:stylesheet id="petstyle" version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="xsl:stylesheet" />

    <xsl:template match="pets">
      My pets:
      <xsl:apply-templates />
    </xsl:template>

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

  </xsl:stylesheet>
</pets>
```

Note

You can run the code in Listing 2.18 only with a processor that supports internal stylesheets. MSXSL and Xalan don't support this (from the command line). With Saxon, you need to use the command-line option `-a`.

ANALYSIS In Listing 2.18, the processing instruction on line 2 links to the internal stylesheet. It refers to the internal stylesheet using the `id` attribute of the stylesheet. This attribute is normally not part of the `stylesheet` element, so you need to explicitly define it. If you don't define it, the processor raises an error. The attribute is defined using a DTD. You don't need to understand the entire syntax of the DTD, but you do need to realize that it refers to the root element of the document it is embedded in—in this case, `pets`.

Another important addition to the embedded stylesheet is a template that matches the stylesheet's root element, `xsl:stylesheet`. This template does nothing but is included to prevent any processing of the embedded stylesheet itself. Otherwise, the stylesheet would be processed by itself and might yield unexpected output.

Executing a Stylesheet Using Code

For most applications using XSLT, the primary way of using a processor is through code targeting the processor's API. Unfortunately, each processor's API is different. In addition, some work with Java, others with Windows COM-based languages, and yet others with other languages and on other platforms. Hence, covering this topic properly would probably take a book of its own and would require a great deal of programming knowledge. To not leave you totally in the dark, I've included Listing 2.19, which shows a sample in VBScript, using the MSXML 3.0 parser/processor component. VBScript is easy to understand, so with a little effort, you should be able to understand how it works.

LISTING 2.19 Transforming XML with XSLT and VBScript

```
Dim objXML, objXSLT
Dim strResult

'Create DOM objects to load XML and XSLT
Set objXML = CreateObject("MSXML2.DOMDocument.3.0")
Set objXSLT = CreateObject("MSXML2.DOMDocument.3.0")

'When loading documents continue after loading has completed
objXML.async = False
objXSLT.async = False

'Load XML and XSLT into objects
objXML.load "02list01.xml"
objXSLT.load "02list11.xsl"

'Transform XML with XSLT and store the result in a string
strResult = objXML.transformNode(objXSLT.documentElement)

```

```
19: 'Release objects
20: Set oXML = Nothing
21: Set oXSLT = Nothing
22:
23: 'Write string to MsgBox
24: MsgBox strResult
```

ANALYSIS If you run Listing 2.19 (`02list19.vbs`) from the command line (in Windows), it searches for the XML and XSLT files specified on lines 13 and 14 and loads them into the XML objects. Line 17 transforms the XML using the provided XSLT object and stores it in the `strResult` string. After all the objects are disposed, the result is written to a message box, which pops up to show the result. Lines 1–2 define all the variables needed in the script. Lines 5–6 create objects to store the XML in (DOM documents), and lines 9–10 set these objects to synchronous mode. This means that the script does not continue until the whole XML (or XSLT) document is loaded. Be aware that the `async` property is not part of the W3C DOM standard but is specific to the MSXML component. Lines preceded with an apostrophe (`'`) are comments. I deliberately wrote Listing 2.19 so that it does not perform error checking to keep the code simple and understandable.

You can run the code in Listing 2.19 from the command line or by double-clicking the `02list19.vbs` file in Windows Explorer. In both cases, you get a dialog box that presents the output. You can't run this code from ASP or other server-side mechanisms. The code works only if the files `02list01.xml` and `02list11.xsl` are in the same folder as `02list19.vbs`. You also need to have MSXML version 3.0 or higher installed. The component's prog ID on lines 5 and 6 is `MSXML2.DOMDocument.3.0`. This version-specific prog ID should work with higher versions as well. If it doesn't, check the component's documentation for the proper value for the program ID.

Summary

Today you learned that a stylesheet's key elements are `xsl:stylesheet` and `xsl:template`. The `xsl:stylesheet` element defines the stylesheet in which templates are used to act on the XML input. Based on matching rules, these templates are fired in such a way that the current element fires only one template. The `xsl:value-of` element writes the value of an element into the output.

Stylesheets can have a simplified syntax that more closely resembles HTML, so people with knowledge of HTML can more easily use XSLT. This simplified syntax does not work with templates, so the capabilities of this syntax are limited.

You can apply stylesheets to an XML document by using a processor, by linking to the stylesheet and having the processor get the linked stylesheet, or by programming with a processor. A linked stylesheet can also be embedded in the XML document itself.

Tomorrow you will learn more details about selecting and outputting data from an XML source document.

Q&A

Q I use a different processor, and the output isn't exactly the same as the results shown in this lesson. Why?

A The W3C specification defines what a processor should do in certain cases. In some cases, the exact behavior is not considered relevant. In those cases, the different processors may yield different results. The differences are related mostly to encoding and whitespace handling.

Q Linking to multiple stylesheets in the prolog looks very useful. How can I use linking?

A Unfortunately, processors are not required to implement stylesheet linking, and only a few do. The ones that do often don't have support for the `alternate`, `media`, and `title` attributes and therefore do not support multiple stylesheets. This support might change in the future if more applications use this feature.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: A stylesheet must contain a prolog with a processing instruction defining the XML version and encoding.
2. True or False: A simplified stylesheet can contain templates.
3. How does a template match an element in the source XML?
4. Which element is needed to get data from elements in the source XML?
5. Why do you need to use the `text()` function to get the text value of an element?

Exercise

1. Create an XML document with the following code:

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   <books>
     <book>Teach Yourself XML in 21 Days</book>
     <book>Teach Yourself XSLT in 21 Days</book>
   </books>
   ```

   Now create an XSLT document that yields the following result if applied to this XML document:

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   My books:
     Teach Yourself XML in 21 Days
     Teach Yourself XSLT in 21 Days
   ```

This page intentionally left blank

DAY 3

Selecting Data

Yesterday you learned what a stylesheet is and how to use it. You also learned about using templates and getting values from an Extensible Markup Language (XML) document. So far the expressions you've used to match templates and select data have been rudimentary. What you can do at this point therefore is limited.

Today's lesson will focus on getting more control over the data you select. Today you will learn the following:

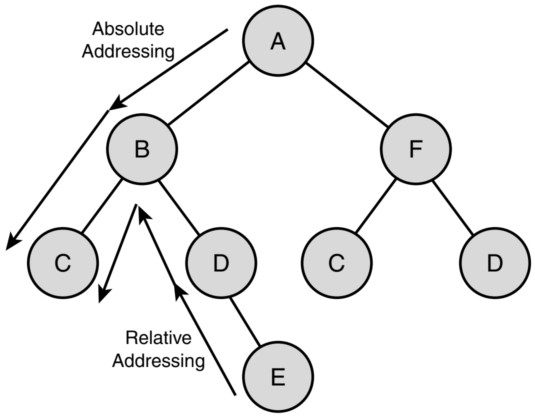
- How the XML document tree works
- What XPath is
- How you can select single elements
- How you can select multiple elements
- How you can select attributes

Understanding the XML Document Tree

An XML document is a hierarchical structure of elements. Each element in an XML document can have zero or more child elements, which in turn have that same property. Also, each element can have zero or more attributes. No surprises so far, but it's actually significant that an XML document is structured this way. Every element and every attribute has a uniquely identifiable place within the document tree. Because all elements and attributes are uniquely identifiable, you can address a single element or attribute and get its value. Figure 3.1 clarifies this structure.

FIGURE 3.1

Graphical representation of a tree.



NEW TERM In Figure 3.1, each element in the tree is shown as a circle. Different children of an element can be distinguished because different letters identify them. As you can see, some elements have children with the same letters to identify them. This means that you can't say "Give me the value of element C," because element C can be the child element of either element B or element F. This doesn't mean, however, that you can't address this element at all; you just have to be more specific. To get the value of a specific element, you would have to say "Give me the value of element C, the child of element B, which is the child of the root element A." When you address an element in this manner, you use absolute addressing, as shown in Figure 3.1. *Absolute addressing* means that you specify the exact location of an element within a tree. With absolute addressing, you always specify a unique location.

NEW TERM Another way of addressing is relative to an element. Say that element E in Figure 3.1 is the element's starting point. If you want to address the same element as before, you can say "Give me the value of element C, the sibling element of my parent element." This type of addressing is called relative addressing, as shown in Figure 3.1. *Relative addressing* means that you specify the location of an element within a tree relative to the position of the current location.

With relative addressing, you don't specify a unique location within the document tree. Which element is specified by the preceding query actually depends on the starting point of the query.

What Is a Node?

Until now, I have been talking about elements and attributes. The difference between elements and attributes is not that great, however. The most important difference is that an element can have child elements; an attribute cannot. Hence, an attribute always has a single (text) value, whereas the value of an element also includes any descendant elements (and attributes).

Because elements and attributes aren't very different, they can be represented as the same thing in a diagram of the XML document tree. Element E in Figure 3.1, for example, could just as well be an attribute because it doesn't have any child elements. In fact, some people think that attributes shouldn't be used because attributes are just special cases of elements. Attributes and elements are interchangeable, as long as an element doesn't have child elements (or attributes). Because attributes are simply names with associated values, also known as name-value pairs, an element can contain only attributes that have different names. An element value, on the other hand, can contain multiple elements with the same name. This distinction is very important when you're designing XML documents, especially when they might have to change in the future.

NEW TERM Within the Document Object Model (DOM), as well as Extensible Stylesheet Language Transformations (XSLT, or actually XPath), the distinction between an element and attribute is so small that they are treated more or less as being the same. Several functions in DOM Level 2 work equally well on elements and attributes. The functions `nodeName` and `nodeValue` make no distinction between elements and attributes, although the result may differ based on the type of node the function is used on. Because an element and an attribute are very similar, they are referred to as a *node,* which is a single item that contains data within the document tree.

Current Node

On Day 2, I used the term current element tentatively. Although this concept is somewhat self-explanatory, some clarification is in order. Also, because of the similarities between elements and attributes, from now on I will use the term *current node*.

NEW TERM On Day 2, you saw that when a stylesheet processes an XML document, elements of the source XML are matched against templates in the stylesheet. What you haven't learned yet is that you can also create match expressions that match an attribute. So, actually, nodes of the source XML are matched against the templates. Each time a match occurs and a template is invoked, the node that fired the template becomes the *current node,* which basically is a pointer to a node within the XML tree. This pointer keeps track of which node is being processed.

Note

If you're working with an XSLT debugger that enables you to perform the transformation process step by step, you can see which node is the current node. The debugger keeps track of the current node and the template that is being fired and shows that information to you.

NEW TERM Because the current node is just a pointer to the node being processed, a template is not limited to accessing the value of that node alone. Within a template, you can use absolute addressing or relative addressing to get the value of any node in the XML document. As I said earlier, this value isn't necessarily a single value. If an element has attributes and descendant elements, they are also part of that value. Such a value is called a *tree fragment,* which is a part of an XML document tree, starting at a specific node. A tree fragment is itself a well-formed XML structure or document.

You already saw tree fragments in action on Day 2, when you learned about the `text()` function that extracts only the text value of an element. If you just specify the value of an element, the text of the element and all its descendants is written to the output. That is, in fact, the text value of the tree fragment. To get a better idea what a tree fragment is, look at Figure 3.2.

FIGURE 3.2
Tree fragment of node T.

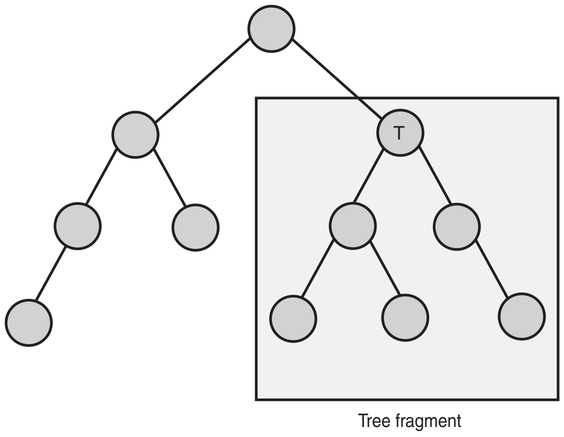


Figure 3.2 is a graphical representation of a tree fragment. In this case, the tree fragment belongs to node T. This is actually the same as the value of node T.

What Is a Node-Set?

Now that you know what a node is, you probably think that a node-set isn't hard to explain. It's a set of nodes, right? Well, yes, but that's not all of it.

New Term When you make a selection based on an expression, the expression doesn't necessarily match one node. It may match several nodes. These nodes together are called a *node-set*. The most common node-set is a series of an element's child nodes. Some people think this is the only kind of node-set, but it isn't. You can easily create an expression that yields a node-set with nodes in different sections of an XML document. Figure 3.3 shows an example.

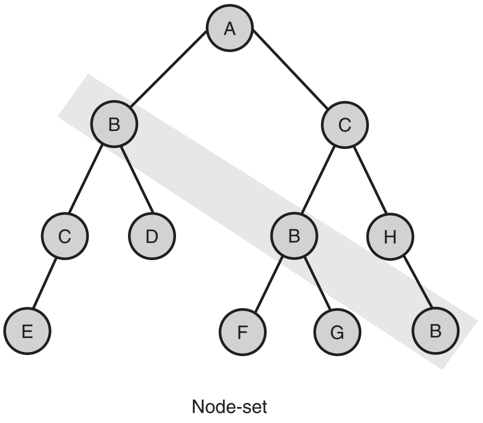


Figure 3.3

Node-set containing nodes scattered throughout an XML document.

Figure 3.3 represents the node-set you would get if you were to say "Give me all nodes named B." As you can see, the node's location in the XML document tree is not relevant. Any node matching your query is part of the node-set. The node-set in Figure 3.3 is composed of several nodes. From those nodes, you have access to the tree fragment composed of that node and its descendants. If the expression targeted only attributes, the node-set would consist of only single value nodes.

Node-sets are essential in XSLT. They enable you to create a table of contents, indexes, and all sorts of other documents in which you use data that is scattered throughout an XML document. This capability enables you to create different outputs for different purposes from the same XML source document.

Understanding XPath

So far, this lesson has been all theory. You need this theory as a foundation for practical application, which is what the rest of this lesson is all about.

XSLT wouldn't work if it didn't have some kind of mechanism to match and select nodes and act on them. You need to be able to express which node or nodes should match. This is what XPath expressions are for. XPath is the language you use to specify which node or nodes you want to work with. Expressions in XPath can be very simple, pointing to a specific location within a document tree using absolute addressing. You can, however, make selections based on very complex rules. As you work your way through this book, you will learn to create more and more complex expressions. But, of course, you need to start simple.

Selecting Elements

If you're familiar with addresses on the World Wide Web, the most basic XPath expressions are easy to understand. A Web site is a hierarchy of files, just like an XML document is a hierarchy of elements. If you visit a Web site, you specify its root with the name of the Web site. For example, `http://www.somesite.com` points to the root or home page of the Web site. This is the same as `http://www.somesite.com/`, which is actually more accurate. What comes after this part of the address specifies where in the hierarchy of the site you want to be. So, `http://www.somesite.com/menu/entrees` points to the index file in the entrees directory, which is a child of the menu directory, which is a child of the root directory. The `/menu/entrees` path is especially interesting. It uniquely identifies a location within the Web site hierarchy, as shown in Figure 3.4.

FIGURE 3.4

Web site hierarchy.

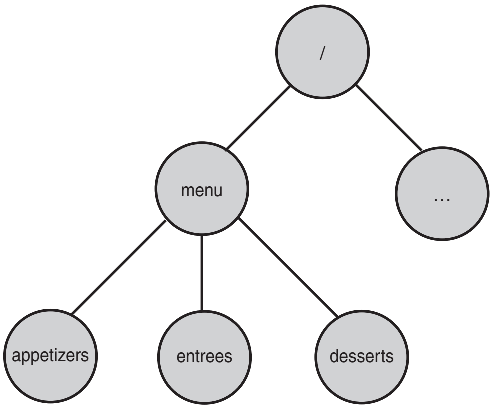


Figure 3.4 shows part of the hierarchy for the Web site. Notice that /menu/entrees uniquely identifies the `entrees` node in the tree. If you want to select the `desserts` node, you change to /menu/desserts. Now look at Listing 3.1.

LISTING 3.1 Menu in XML Corresponding to Figure 3.4

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
   <dish id="10" price="6.95">Dame Blanche</dish>
   <dish id="11" price="5.95">Chocolat Mousse</dish>
   <dish id="12" price="6.95">Banana Split</dish>
 </desserts>
</menu>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS The XML in Listing 3.1 has the same tree structure as that of the Web site depicted in Figure 3.4. So, just like in the Web site, `/menu/entrees` points to the `entrees` element in the XML document. Pointing to a certain node in an XML document with XPath is, as you can see, very simple. It is based on principles that you have probably used before, so they'll be familiar to you, even though you've never worked with XPath before. To see how this approach really works, look at Listing 3.2.

LISTING 3.2 Stylesheet Selecting the `entrees` Node from Listing 3.1

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:value-of select="/menu/entrees" />
7:    </xsl:template>
8:
9:  </xsl:stylesheet>
```

ANALYSIS The template on line 5 matches the root element of the source document. The value retrieved on line 6 is selected using the expression `/menu/entrees`, which matched the `entrees` element that is the child element of the root element `menu`. The result from applying Listing 3.2 to Listing 3.1 is shown in Listing 3.3.

OUTPUT **LISTING 3.3** Result from Applying Listing 3.2 to Listing 3.1

```
<?xml version="1.0" encoding="utf-8"?>
    Grilled Salmon
    Seafood Pasta
    Linguini al Pesto
    Rack of Lamb
    Ribs and Wings
```

Note

Be aware that the preceding sample was run with the Saxon processor. If you use another processor, the result might be slightly different. MSXSL generates UTF-16 output by default, so the result when using MSXSL will have spaces between each letter.

ANALYSIS Listing 3.3 shows the value of all the child nodes of the `entrees` node. If you remember yesterday's lesson, that is exactly right, as line 6 of Listing 3.2 asks for the value of the `entrees` node. That node's value contains all its descendant nodes. Getting its value yields the text value of the descendant elements. This scenario is a bit confusing because it looks like Listing 3.2 actually selects a node-set consisting of all the child elements of the `entrees` node. If the `entrees` node also contains a text value, you would see that this isn't true. You can, however, create an additional template to handle the `dish` elements, as shown in Listing 3.4.

LISTING 3.4 Stylesheet with More Control over `dish` Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="/menu/appetizers" />

  <xsl:template match="/menu/entrees">
    Entrees:
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="/menu/desserts" />

  <xsl:template match="dish">
```

```
19:     -<xsl:value-of select="text()" />
20:   </xsl:template>
21:
22: </xsl:stylesheet>
```

ANALYSIS In Listing 3.4, note that lines 9 and 16 effectively ignore the `appetizers` and `desserts` nodes in Listing 3.1 to keep the result small and to the point. The result of applying Listing 3.4 to Listing 3.1 is shown in Listing 3.5.

OUTPUT **LISTING 3.5** Result from Applying Listing 3.4 to Listing 3.1

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


     Entrees:


    -Grilled Salmon

    -Seafood Pasta

    -Linguini al Pesto

    -Rack of Lamb

    -Ribs and Wings
```

ANALYSIS In Listing 3.5, each `dish` node is now handled separately. The hyphen (`-`) in front of each dish shows that this is really the case. The whitespace appears, as I said before, because of the processor's default whitespace rules.

Getting the Value of a Single Element

The problem with the code shown so far is that it acts on an element or a set of elements. Within the set of elements (such as the `dish` elements), no one node is singled out. If you also address a `dish` node instead of matching it with a template, a reasonable assumption would be that you will get the value of a single `dish` node. Listing 3.6 tests this assumption.

LISTING 3.6 Stylesheet Getting the Value of a `dish` Element

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

```

LISTING 3.6 Continued

```
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="/menu/appetizers" />
10:
11:   <xsl:template match="/menu/entrees">
12:     <xsl:value-of select="dish" />
13:   </xsl:template>
14:
15:   <xsl:template match="/menu/desserts" />
16:
17: </xsl:stylesheet>
```

ANALYSIS In Listing 3.6, the template on line 11 matching the `entrees` element selects only the value of a `dish` node on line 12. Note that, compared to Listing 3.4, there is no template matching the `dish` element. The result for Listing 3.6 is shown in Listing 3.7.

OUTPUT **LISTING 3.7** Result from Applying Listing 3.6 to Listing 3.1

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

  Grilled Salmon
```

ANALYSIS In Listing 3.7, the assumption made for Listing 3.6 is correct. Getting the value of a `dish` node yields the value of exactly one `dish` node. But what happened to the other nodes? After all, the `xsl:value-of` element matches `dish` nodes, so it, in fact, matches a node-set. The fact that you get a single value is again a result of the default behavior of XSLT. If a node-set matches a `xsl:value-of` selection, only the value of the first element in the node-set is used. The first element is the one that comes first in the source XML, according to the selection—in this case, the Grilled Salmon.

Now a new question arises: How do you specifically select another element in the node-set? Fortunately, you can just specify the number of the element you want to select. This, however, deviates from the path notation you are familiar with from Web sites. You need to place the number of the element you want to select between square brackets, [and]. Hence, you select the third `dish` element like this:

```
<xsl:value-of select="dish[3]" />
```

Note

> In many programming languages, a list of elements is numbered from 0, so element number 3 is actually the fourth element in the list. XSLT numbers a list starting with 1, so element number 3 is the third element.

New Term The value between the square brackets is called a *predicate,* which is a literal or expression that determines whether a certain node should be included in the selection.

Note that the preceding example uses relative addressing, which means that the selection is done based on the current location. If that is the `entrees` element, the third dish element in the `entrees` element is selected. If the current node has no child elements named `dish`, the value is empty.

Because predicates can be expressions, they can become quite complex, testing whether an element conforms to certain criteria. As I said at the beginning of this section, I'll discuss more complex expressions later in this book. The object now is to make you familiar with the building blocks, so let's proceed with an example based on what you've seen so far. The example in Listing 3.8 creates a menu of the day from the sample XML in Listing 3.1.

Listing 3.8 Stylesheet Creating "Today's Menu"

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      Today's Menu
7:      -<xsl:value-of select="/menu/appetizers/dish[2]" />
8:      -<xsl:value-of select="/menu/entrees/dish[3]" />
9:      -<xsl:value-of select="/menu/desserts/dish[1]" />
10:   </xsl:template>
11:
12: </xsl:stylesheet>
```

Analysis Listing 3.8 has only one template on line 5, matching the root element. This template displays the values for different elements in Listing 3.1 using absolute addressing and number predicates. Line 7 selects the second `dish` element in the `appetizers` element; line 8, the third `dish` element in the `entrees` element; and line 9, the first `dish` element in the `desserts` element. Listing 3.9 shows the result.

OUTPUT **LISTING 3.9** Result from Applying Listing 3.8 to Listing 3.1

```
<?xml version="1.0" encoding="utf-8"?>
     Today's Menu
     -Jumbo Prawns
     -Linguini al Pesto
     -Dame Blanche
```

ANALYSIS Listing 3.8 contains only a template matching the root of the XML source. The rest of the stylesheet's functionality utilizes absolute addressing to get the wanted values. As you can see, this yields a list of dishes that form today's menu.

NEW TERM Earlier, you learned about the current node. After a template is fired, a certain node is considered the current node. A path expression doesn't contain a current node, but it does consist of context nodes. A *context node* is a part of an expression that operates on a certain node.

The predicates used in Listing 3.8 operate on the context node—in this case, the `dish` node. At one point or another, each part of the path's expression is the context node. It, however, is relevant only when you're working with predicates in a path expression. You can have predicates at several stages within the path expression, in which case the context node is the node that the predicate operates on.

You must realize that if no match occurs, nothing happens. This is also the case if a predicate holds a number for which no element exists. In that case, the number is just ignored. You don't see an error message or anything telling you that an element is missing. The clue is not that an element is missing, but rather that no element matches that particular rule, so the rule is never applied.

Selecting Attributes

So far, you've learned only about elements. But what about attributes? Earlier, I said that elements and attributes don't differ very much, so you might expect that you can address them in the same way, as Listing 3.10 tries to do.

LISTING 3.10 Stylesheet Trying to Select Attributes

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      Dessert of the Day:
7:      <xsl:value-of select="/menu/desserts/dish[2]" />
8:      Price: <xsl:value-of select="/menu/desserts/dish[2]/price" />
```

```
9:    </xsl:template>
10:
11: </xsl:stylesheet>
```

ANALYSIS Line 8 in Listing 3.10 tries to get the value of the `price` attribute of a `dish` element. That this approach doesn't work is obvious from the result in Listing 3.11.

OUTPUT **LISTING 3.11** Result from Applying Listing 3.10 to Listing 3.1

```
<?xml version="1.0" encoding="utf-8"?>
     Dessert of the Day:
     Chocolat Mousse
     Price:
```

ANALYSIS The `value-of` element in Listing 3.10 doesn't yield a result. A gap appears in the result because no element matches the `select` expression. That is as it should be, because no `price` element exists. The `dish` element does, however, have a `price` attribute, which is what Listing 3.10 is supposed to select. What's wrong?

Nothing is wrong. You just need to tell the processor that it needs to match an attribute, not an element. You can tell the processor that you're looking for an attribute by adding the `@` character in front of the name. So, if line 8 in Listing 3.10 is supposed to point to an attribute, the path expression should be `/menu/desserts/dish[2]/@price`, as shown in Listing 3.12. The result in Listing 3.13 is now correct.

LISTING 3.12 Stylesheet Correctly Selecting an Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    Dessert of the Day:
    <xsl:value-of select="/menu/desserts/dish[2]" />
    Price: <xsl:value-of select="/menu/desserts/dish[2]/@price" />
  </xsl:template>

</xsl:stylesheet>
```

OUTPUT **LISTING 3.13** Result from Applying Listing 3.12 to Listing 3.1

```
<?xml version="1.0" encoding="utf-8"?>
     Dessert of the Day:
     Chocolat Mousse
     Price: 5.95
```

Analysis Listing 3.12 produces the desired result. Because attributes don't have any child elements, you also know that no side effects will occur. The value of the attribute is always text.

Note Attributes can have data types, but all are based on the text value. Data types will be thoroughly discussed on Day 10, "Working with Data Types."

Because attributes don't have side effects, they are much easier to work with. Also, because all attributes of an element need to have a different name, you have no trouble with matching multiple attributes (and getting only the value of the first). The only way you can get multiple attributes is to start working with selections based on a wildcard character.

Another point to consider is that attributes take less space in a document than elements. Elements have begin and end tags; an attribute doesn't need these tags. Because attributes have only a text value, a parser or processor can deal with them more quickly because it doesn't have to check for child elements. This is likely to have a positive effect on performance.

Beyond the Basics

Until now, the discussion has targeted single nodes wherever possible. In fact, the focus has been on how to avoid selections that yield more than one node. Although this information is very useful to get you started, it really limits your capabilities. Without creating complex expressions, you can already perform many tasks with some of the basic functionality XPath provides.

Using a Wildcard

Wildcard characters are common in most search-oriented functions and languages. XPath has only one wildcard character: `*`. You can use it only to match entire names of elements or attributes, so the expression `a*` does not match all elements starting with the letter *a*. This expression generates an error instead. Wildcards are useful when you want to drill deeper into the source XML, and the names of certain nodes (particularly parent nodes) don't matter. Listing 3.14 shows how to use a wildcard.

Listing 3.14 Stylesheet Using a Wildcard

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
```

```
5:    <xsl:template match="/menu/*/dish">
6:      -<xsl:value-of select="text()" />
7:    </xsl:template>
8:
9:  </xsl:stylesheet>
```

ANALYSIS Line 5 in Listing 3.14 uses a wildcard character, so it doesn't matter whether the matched `dish` element is a child element of the `appetizers`, `entrees`, or `desserts` element. Line 6 just shows the value of the matched `dish` element. Listing 3.15 shows the result.

OUTPUT **LISTING 3.15** Result from Applying Listing 3.14 to Listing 3.1

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


     -Crab Cakes

     -Jumbo Prawns

     -Smoked Salmon and Avocado Quesadilla

     -Caesar Salad


     -Grilled Salmon

     -Seafood Pasta

     -Linguini al Pesto

     -Rack of Lamb

     -Ribs and Wings


     -Dame Blanche

     -Chocolat Mousse

     -Banana Split
```

ANALYSIS In Listing 3.15, the result yields all the `dish` nodes, not just those that are child nodes of a particular node.

You can use this technique in all kinds of situations. Say that you've created a white-paper or book using an XML document. Using a wildcard, you can select all the chapter and section headers to create a table of contents. If you don't want to get all the nodes, using just the wildcard doesn't solve your problem. However, just as with the path expressions you saw earlier, you can use predicates to refine the expression. That way, you have more control over what the wildcard actually matches. A simple example is shown in Listing 3.16.

LISTING 3.16 Stylesheet Using Wildcards and Predicates

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:   <xsl:template match="/">
6:      Dessert of the Day:
7:      <xsl:value-of select="/menu/*[3]/dish[2]" />
8:      Price: <xsl:value-of select="/menu/*[3]/dish[2]/@*[2]" />
9:   </xsl:template>
10:
11:  </xsl:stylesheet>
```

ANALYSIS Listing 3.16 yields the same result as Listing 3.12; the result is shown in Listing 3.13. Instead of addressing nodes by name, line 8 in Listing 3.16 uses several wildcards aided by position predicates. The `desserts` element is the third child element of the `menu` element. The `/menu/*[3]` section of the path expression tells the processor to take the third child element of the `menu` element, with no regard to the name of that element. That expression yields the `desserts` element, just as if it had been named. The attribute chosen is also based on a wildcard. In this case, the expression tells the processor to take the second attribute of the `dish` element, which is, of course, the `price` attribute.

Working with the Document Tree

NEW TERM What you've seen so far is quite rigid in utilizing the XML document tree. XPath can do much more, based on *location paths*, which are expressions that select a node or node-set relative to the context node. A location path consists of several parts. The path expressions you've seen so far often are specific cases of location paths, starting from the root node or current node (from a matching template). Location paths can, however, appear in other instances and as part of a predicate within a path expression. One part of a location path is actually a predicate, so you can have a location path with a predicate, containing a location path containing another predicate, and so on.

NEW TERM Another part of a location path that you are already familiar with is called a *node test,* which is the part of the location path that matches a certain node or nodes. This definition is clearer with an example. Consider the following location path:

```
/menu/*[3]/dish[2]/@*[2]
```

Here, `menu`, `*`, `dish`, and `@*` are node tests.

New Term The last (or actually the first) part of a location path may not be familiar to you yet. This part, called an *axis,* is an expression specifying the relationship within the document tree between the selected nodes and the context node.

In the previous examples, you saw quite a few location paths. Not all of them contained axes, however. Well, they did, but the axes were included implicitly. Look at the following location path:

```
/menu/desserts/dish
```

This location path selects all the `dish` elements, which are child nodes of `desserts` nodes, which in turn are child nodes of the root element `menu`. If you write out that location path in full, it actually reads

```
/child::menu/child::desserts/child::dish
```

The axis and the node test are always separated by a double colon. The axis in front of the node test tells the processor where to look for a node or node-set. The node test tells the processor which nodes to actually match. Using the explicit location path isn't very useful for match templates. The location path only becomes lengthier and less readable. Also, you would not often use the `child` axis because it is included implicitly anyway.

Another axis you are already familiar with is the attribute axis. You are actually familiar with its shortcut, the `@` character. In Listing 3.12, the following location path was used on line 8:

```
/menu/desserts/dish[2]/@price
```

You also can write the `@price` selection using the attribute axis, which would yield the following location path:

```
/menu/desserts/dish[2]/attribute::price
```

An axis that you haven't yet encountered but is quite clear is the `parent` axis. Yes, you guessed it: It returns the parent node of the current context. Listing 3.17 shows an example using the `parent` axis.

Listing 3.17 Stylesheet Using `parent` Axis

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <xsl:stylesheet version="1.0"
3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
```

LISTING 3.17 Continued

```
5:    <xsl:template match="/"><xsl:apply-templates /></xsl:template>
6:
7:    <xsl:template match="/menu/appetizers/dish">
8:     <xsl:value-of select="parent::*/@title" />: <xsl:value-of select="." />
9:    </xsl:template>
10:
11:   <xsl:template match="/menu/entrees" />
12:   <xsl:template match="/menu/desserts" />
13:
14:  </xsl:stylesheet>
```

ANALYSIS Line 8 in Listing 3.17 uses the `parent` axis to select the `title` attribute of the parent element. This is the parent element of the current node, which is one of the `dish` elements in the `appetizers` element. Listing 3.18 shows the result.

OUTPUT **LISTING 3.18** Result from Applying Listing 3.17 to Listing 3.1

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

    Work up an Appetite: Crab Cakes
    Work up an Appetite: Jumbo Prawns
    Work up an Appetite: Smoked Salmon and Avocado Quesadilla
    Work up an Appetite: Caesar Salad
```

ANALYSIS In Listing 3.18, the `title` attribute is inserted with every dish. This could also have been done with absolute addressing, but if there had been a separate template for the `dish` element, dishes in `entrees` or `desserts` could also match and the title would need to be different. In that case, relative addressing using the `parent` axis would be the only way out.

There is another way to specify *any* parent node. If you're familiar with the command prompt from DOS or Unix, it will look familiar. The location path

```
parent::*/@title
```

can also be written as

```
../@title
```

The latter example is much more compact and usable. If, however, the node test is not a wildcard, using this location path would not work. You would have to specify the `parent` axis explicitly, as well as the node that the node test needs to match.

Multilevel Axes

The axes discussed so far cover a single level in the XML document tree. They either go one level down, to child nodes, or one level up, to the parent node. Several axes operate on multiple levels. This does not mean, however, that such an axis returns a tree fragment because that wouldn't give added functionality. If that were the case, the parent and child axes would suffice. Each multilevel axis yields a node-set containing all the elements within that axis. The order in which the elements appear in the node-set depends on the axis. So, you can actually think of a multilevel axis as part of the XML document tree "flattened" into a set.

The best way to show you how these axes work is to go through some examples. The easiest axis to start with is ancestor, which is shown in Listing 3.19.

LISTING 3.19 Stylesheet Using the Ancestor Axis

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/menu/appetizers">
6:      -<xsl:value-of select="dish[1]/ancestor::*[1]/@title" />
7:      -<xsl:value-of select="dish[1]/ancestor::menu/entrees/@title" />
8:    </xsl:template>
9:
10:   <xsl:template match="/menu/entrees">
11:   <xsl:template match="/menu/desserts">
12:
13: </xsl:stylesheet>
```

ANALYSIS All the action occurs in Lines 6 and 7 in Listing 3.19. Both lines first select the first child node of the `appetizers` element named `dish`. At that point, this element becomes the context node. Next, the ancestor axis tells the processor it wants to act on the ancestor nodes of the context node. The ancestor nodes are `appetizers` and `menu`. On line 6, the node test consists of a wildcard and predicate—in this case, pointing to the first ancestor node in the axis, which is the parent node `appetizers`. On line 7, a named node test specifies that the `menu` element is the element needed in this particular axis. This line could also have been written as `ancestor::*[2]` because that is the grandparent of the context node and, as such, the second node in the axis node-set ancestor.

OUTPUT **LISTING 3.20** Result from Applying Listing 3.19 to Listing 3.1

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

       -Work up an Appetite
       -Chow Time!
```

ANALYSIS Listing 3.20 shows that Listing 3.19 first selects the `title` attribute of the `appetizers` element and then the `title` attribute of the `entrees` element.

Several more multilevel axes are available. All axes are listed in Table 3.1.

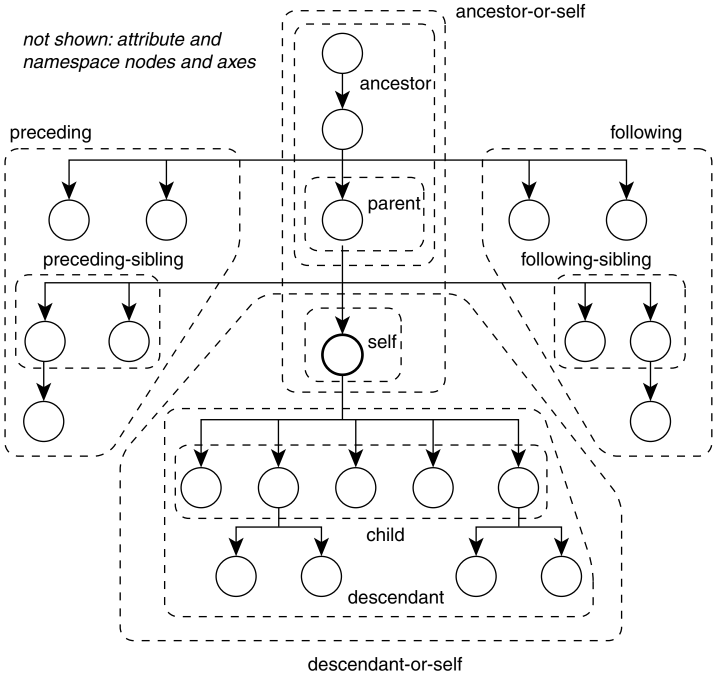
TABLE 3.1 Available Axes

Axis	*Description*
`Self`	The context node itself
`child`	All child nodes of the context node
`parent`	The parent node of the context node
`ancestor`	All ancestor nodes of the context node
`ancestor-or-self`	Same as `ancestor`, including the context node as the first node in the node-set
`descendant`	All descendant nodes of the context node, numbered depth first (for example, child 1, grandchild 1, child 2, grandchild 2, and so on)
`descendant-or-self`	Same as `descendant`, including the context node as the first node in the node-set
`following-sibling`	All sibling nodes that succeed the context node within the document tree
`following`	Same as `following-sibling`, but including their descendants, depth first (for example, sibling 1, child 1 of sibling 1, sibling 2, and so on)
`preceding-sibling`	All sibling nodes that precede the context node within the document tree
`preceding`	Same as `preceding-sibling`, but including their descendants, depth first (for example, sibling 1, child 1 of sibling 1, sibling 2, and so on)

Figure 3.5 shows a graphical representation of most axes in Table 3.1 to give you a better idea of what each axis listed actually selects.

FIGURE 3.5

Graphical representation of the axes in Table 3.1.



Something that is, strictly speaking, not an axis but fits nicely in this section is the `//` expression. It matches any location within the document tree. So, `//dish` matches any `dish` node within the document. Listing 3.21 shows this expression in action.

LISTING 3.21 Stylesheet Using `//` to Get All Nodes

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="//dish">
6:      -<xsl:value-of select="text()" />
7:    </xsl:template>
8:
9:  </xsl:stylesheet>
```

ANALYSIS Listing 3.21 yields the same result as Listing 3.14 because line 5 uses `//` to match all nodes in the document from which the `dish` nodes are selected.

A common mistake is to think that `//dish[6]` will yield the sixth dish element within the document. Unfortunately, because of precedence rules, this expression will yield no element at all. The processor will look for a sixth element in each of the parent contexts. Unfortunately, there is none because `appetizers` contains only four `dish` elements; `entrees`, only five; and `desserts`, only three. So how do you solve this problem? You can use parentheses to override the default precedence. Therefore, `(//dish)[6]` would yield the sixth element (the Seafood Pasta).

Summary

Today you learned that elements and attributes alike are referred to as nodes. You also learned that a node with child nodes (elements) is called a tree fragment. Using XPath expressions, you can also match nodes in different parts of an XML document. If the match expression matches more than one node, the result is a node-set in which each node can be referred to by number within the node-set. A node-set is not a hierarchy of nodes.

You select nodes by using XPath expressions. They consist of location paths, which in turn consist of an axis, a node test, and a predicate. The axis and predicate are optional. If no axis is specified, it is implied. If no predicate is specified, all nodes that match the axis and node test will match.

Tomorrow you will learn more information about templates. You will use many of the rules you learned today about XPath expressions as you learn more about templates and controlling the flow of control.

Q&A

Q How can I tell if an expression will match a node or a node-set?

A Unless a predicate specifically targets a specific node or the expression matches an attribute, you can't tell whether your expression will match a node or a node-set. This is by design because it actually shouldn't matter whether you are matching a node or a node-set; each node matching the expression is supposed to be processed. You can download a tool called XPath Visualizer from `http://www.topxml.com/xpathvisualizer/default.asp`. This tool can help you determine whether your expression is correct.

Q How can I tell whether an expression in a `value-of` command will yield a text value or a tree fragment?

A You have no way of knowing whether you are getting a text value or a tree fragment unless you use the `text()` function.

Q I have a stylesheet that keeps outputting text that I didn't ask for. How can I get rid of this problem?

A The default behavior of the stylesheet causes this problem. Check whether all nodes are being matched by a template. Also, check that your expression outputting values doesn't match a tree fragment.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: The `ancestor-or-self` axis includes the context node.
2. True or False: A predicate can contain a location path.
3. Given Listing 3.1, what would be the output of `<xsl:value-of select="/menu/entrees/dish" />`?
4. Given Listing 3.1, what would be the output of `<xsl:value-of select="//dish[10]/parent::*/@title" />`?
5. Given Listing 3.1, what would be the output of `<xsl:value-of select="(//dish)[10]/parent::*/@title" />`?

Exercise

1. Create a stylesheet that displays Listing 3.1 nicely as a menu with sections, using the `title` attribute as a header for each section.

This page intentionally left blank

DAY 4

Using Templates

Yesterday you learned how to select element attributes from Extensible Markup Language (XML) source with XPath expressions. By using these expressions when matching templates, you achieved more control over the output from a stylesheet applied to an XML source. XPath expression templates form the cornerstone of XSLT. So, now it's time to learn more about templates and what you can do with them.

In today's lesson covering option templates, you will learn the following:

- Why templates are so significant in XSLT
- How to work with match templates
- How to work with named templates
- What happens when more than one template matches
- How to give priorities to templates

Understanding Templates

Templates are a key concept in XSLT. If you don't fully understand them, you cannot fully understand and utilize the power of XSLT. To make sure that you know what templates are all about and how to use them, today's lesson is all about templates.

A Closer Look at Templates

You have already performed some tasks using templates, so by now you should understand what a template is and does. The next step is to build on the practical basics you know to get a deeper understanding of the template's role.

Back to the Beginning

Before you move on to new topics, you need to go back to the basics described already and formalize your understanding.

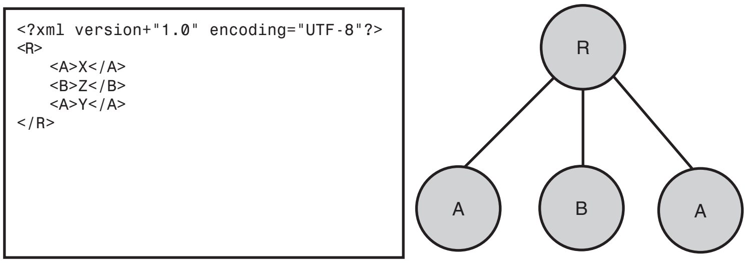
A template contains a set of instructions that are executed when the template is called. These instructions are other XSLT elements. A template either can be called explicitly or can be matched based on a rule against a node in the XML document tree being processed. Explicit calls will be discussed later in this lesson.

A template can be called based on a matching node in the XML document tree if it contains a `match` attribute. The `match` attribute's value should be an XPath expression that describes which nodes in the XML document tree the template applies to.

The fact that nodes are matched rather than selected is an important difference between XSLT and programming languages such as C, Java, and Visual Basic. When nodes are matched, they are processed as the processor encounters them in the XML document tree. This means that if two nodes matched by the same template are not in sequence, they are not processed in sequence. Any nodes in between, possibly matched by another template, are processed in between. This sequence of execution is different from most programming languages, in which you specify on which data you want to operate and then that data is processed in sequence, regardless of its position within the data store it came from. Figure 4.1 helps to illustrate this operation.

FIGURE 4.1

XML document and its tree representation.



```
<?xml version+"1.0" encoding="UTF-8"?>
<R>
   <A>X</A>
   <B>Z</B>
   <A>Y</A>
</R>
```

The document (and hence the document tree) in Figure 4.1 is simple. Basically, this document contains two nodes named A, with one node named B in between. Now look at Listing 4.1.

LISTING 4.1 Partial Stylesheet for Figure 4.1

```
1:  <xsl:template match="A">
2:    <xsl:value-of select="." ?>
3:  </xsl:template>
4:
5:  <xsl:template match="B">
6:    <xsl:value-of select="." ?>
7:  </xsl:template>
```

ANALYSIS Listing 4.1 is a partial listing of a stylesheet containing two templates: one matching nodes named A on line 1 and the other matching nodes named B on line 5. Because of the XML document's structure, the output of these templates is as shown in Listing 4.2.

OUTPUT **LISTING 4.2** Output from Listing 4.1 Applied to Figure 4.1

```
x
z
y
```

ANALYSIS Listing 4.2 is only a partial output (the stylesheet in Listing 4.1 is also only partially shown), but you can see that the first template is matched, then the second, and then the first again. The sequence these templates appear in the stylesheet doesn't matter; the output is the same. Because XSLT is data driven, the data determines which template is executed. So, the templates are executed in the order the data appears in the XML source document, not the order in which the templates appear in the stylesheet.

Templates Explained

Elements within a template are executed in the order in which they appear in the template. The elements in the template form a structure in which the data is inserted in key places. That's why templates are called templates; that's what they are. Templates are there to format or reformat data, so data can be displayed in an eye-pleasing manner or transformed into another data format.

In a simplistic view, XSLT is more or less like Cascading Stylesheets (CSS). Any HTML element for which a CSS style is defined is displayed in that style—for instance, bold red text in the Verdana typeface. XSLT, however, goes much further than just applying styling information; it defines a structure in which the data is inserted. In this sense, XSLT can be better compared to a mold. In a mold, you pour a liquid that solidifies in the shape of the mold; in an XSLT template, you put data that is outputted in the shape defined by the template. The essence of the two is the same: You put in something that has no predetermined shape and can therefore be shaped into anything the material is capable of, and you get out something that is shaped the way you want it. After it is shaped, getting back the original shapeable form is probably very hard. If you transform XML into HTML, the meaning of the data gets lost in the process and is replaced by formatting data. Getting back data with even a similar meaning is very tricky, if not impossible.

The Benefit of Using Templates

XSLT is all about displaying or transforming data. If you didn't have templates, the code for making the changes you want would probably be very long. When you use templates, this code is shortened dramatically, as the templates are reused for any node matching the match expression. The nodes matching an expression are not limited to nodes with the same names. Expressions can be very complex and match a multitude of nodes, both elements and attributes. Because you use just one expression, this is a very powerful way of reusing code. In languages such as C, Java, and Visual Basic, you would have to write a series of statements for each node type that should match and explicitly call a procedure or function to deal with it. In XSLT, the processor performs all this work for you—a further reminder that XSLT is a declarative programming language.

Targeting Multiple Documents with One Stylesheet

Using templates, you can create stylesheets that can act on many different XML documents, even if they have different structures. Being able to do so doesn't mean that you can process *all* XML documents with one stylesheet, but the structure of the data in a document doesn't necessarily have to be a one-to-one match with data from another document. The nodes that are the same in both name and structure will, however, come out the same for each document. An example will show this point clearly. Listing 4.3 shows the stylesheet you can use.

LISTING 4.3 Stylesheet Handling Different Documents

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
```

```
8:
9:    <xsl:template match="employee">
10:     <xsl:value-of select="@name" />
11:     <xsl:apply-templates />
12:   </xsl:template>
13:
14:   <xsl:template match="car">
15:     Manufacturer: <xsl:value-of select="@manufacturer" />
16:     Model:        <xsl:value-of select="@model" />
17:     Year:         <xsl:value-of select="@year" />
18:     Plate:        <xsl:value-of select="@plate" />
19:     <xsl:apply-templates />
20:   </xsl:template>
21:
22:   <xsl:template match="parkinglot">
23:     Parking Lot:  <xsl:value-of select="@number" />
24:   </xsl:template>
25: </xsl:stylesheet>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS Listing 4.3 contains several templates, operating on different elements. The template on line 5 matches the root of the source document and immediately continues processing so that other templates are matched. The template on line 9 matches any `employee` element, outputs a value on line 10, and invokes templates for any child elements on line 11. The template on line 14 matching any `car` element is not much different but outputs several attribute values, and then on line 19 the processing is continued again. The last template on line 22 outputs only the value of the `number` attribute. It does not invoke other templates. Listing 4.4 shows a possible input for this stylesheet.

LISTING 4.4 Sample XML Listing to Be Used with Listing 4.3

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

ANALYSIS Listing 4.4 shows a list of cars. Each car element has a `model`, `manufacturer`, and `year` attribute with different values for each element. If you apply the stylesheet in Listing 4.3 to it, the result is similar to Listing 4.5.

OUTPUT **LISTING 4.5** Result from Applying Listing 4.3 to Listing 4.4

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

    Manufacturer: Ford
    Model:        Focus
    Year:         2000
    Plate:

    Manufacturer: Volkswagen
    Model:        Golf
    Year:         1999
    Plate:

    Manufacturer: Toyota
    Model:        Camry
    Year:         1999
    Plate:

    Manufacturer: Honda
    Model:        Civic
    Year:         2000
    Plate:

    Manufacturer: Chevrolet
    Model:        Prizm
    Year:         2000
    Plate:
```

ANALYSIS The output from Listing 4.3 applied to Listing 4.4 is a nicely formatted list of cars. The template matching the `car` element is responsible for this formatting. Note that for each car in the output, no plate is listed because the elements in the source XML don't have a `plate` attribute. Also, there is no output from the `employee` and `parkinglot` templates. No data matches them, so they aren't called, and thus do not produce any output.

Listing 4.6 shows a different XML document that you can use with Listing 4.3.

LISTING 4.6 Sample XML Listing to Be Used with Listing 4.3

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <employee name="Joe Dishwasher">
```

```
    <car model="Mustang" manufacturer="Ford" year="1981" plate="UDM988">
      <parkinglot number="17" />
    </car>
  </employee>
  <employee name="Carol Waitress">
    <car model="Metro" manufacturer="Geo" year="1997" plate="CDX236">
      <parkinglot number="7" />
    </car>
  </employee>
</employees>
```

ANALYSIS The structure of the document in Listing 4.6 is different from Listing 4.4. It lists employees by name and adds information about the cars they have and which parking lot numbers they have. The `car` element is similar to that of Listing 4.4, but it adds an attribute for the license plate. Listing 4.7 shows the result when the stylesheet from Listing 4.3 (previously used with Listing 4.4) is used with Listing 4.6.

OUTPUT **LISTING 4.7** Result from Applying Listing 4.3 to Listing 4.6

```
<?xml version="1.0" encoding="utf-8"?>
  Joe Dishwasher

    Manufacturer: Ford
    Model:        Mustang
    Year:         1981
    Plate:        UDM988

    Parking Lot:  17


  Carol Waitress

    Manufacturer: Geo
    Model:        Metro
    Year:         1997
    Plate:        CDX236

    Parking Lot:  7
```

ANALYSIS The result in Listing 4.7 is somewhat different. The information regarding the cars, however, is displayed in the same way in both Listings 4.5 and 4.7. The same template is matched for those elements, although the position of the `car` elements within the XML document is entirely different. With information added about the employees and their parking lots, the templates operating on those elements are fired and this time around produce output. Also, now that the `plate` attribute has been added, note that the car information in the output also contains the values of the license plates.

The preceding samples give you a good idea of how you can benefit from templates as reusable code across different data structures. The nice thing is that when you need to differentiate between the same element in differently structured data, you can. The match expressions created with XPath enable you to match with more precision if necessary, as shown in Listing 4.8.

LISTING 4.8 Partial Stylesheet with Templates Handling the Same Element with Different Parents

```
<xsl:template match="cars/car">
  Manufacturer: <xsl:value-of select="@manufacturer" />
  Model:        <xsl:value-of select="@model" />
  Year:         <xsl:value-of select="@year" />
</xsl:template>

<xsl:template match="employee/car">
  Manufacturer: <xsl:value-of select="@manufacturer" />
  Model:        <xsl:value-of select="@model" />
  Year:         <xsl:value-of select="@year" />
  Plate:        <xsl:value-of select="@plate" />
  <xsl:apply-templates />
</xsl:template>
```

ANALYSIS Listing 4.8 shows two templates to replace the `car` template on lines 14 through 20 in Listing 4.3. These templates make a difference between `car` elements that are child nodes of a `cars` element and of an `employee` element. The template handling the latter hasn't really changed, so the output would remain the same. The former, however, no longer tries to output the value of the `plate` attribute, so the result looks like Listing 4.9.

OUTPUT **LISTING 4.9** Result from Applying Listing 4.8 to Listing 4.4

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

    Manufacturer: Ford
    Model:        Focus
    Year:         2000

    Manufacturer: Volkswagen
    Model:        Golf
    Year:         1999

    Manufacturer: Toyota
    Model:        Camry
    Year:         1999
```

```
Manufacturer: Honda
Model:        Civic
Year:         2000

Manufacturer: Chevrolet
Model:        Prizm
Year:         2000
```

ANALYSIS The result in Listing 4.9 is different because it no longer outputs the text `Plate:` with each car. Listing 4.8 no longer outputs this text, and processing of the element is stopped after the value of the `year` attribute is displayed. The processing stops because the template no longer uses `apply-templates` to invoke new templates.

Working with Ad Hoc Data

The example in the preceding section is useful if you're working with data that is structured somewhat ad hoc. A nice example is article text or something that contains headers, quotes, keywords, and so on tagged using XML. This document is much like HTML, but the tags tell what something is, not how to display it. The example in Listing 4.10 shows part of such a document.

LISTING 4.10 Partial Document of XML Tagged Text

```
<analysis>
Listing 3.3 shows the value of all the child nodes of the <code>entrees</code>
node. If you remember yesterday's lesson that is exactly right, as Listing 3.2
(line 6) asks for the value of the <code>entrees</code> node. The value of
that node contains all its descendant nodes. Getting its value yields the text
value of the <keyword>descendant</keyword> elements. This is a bit confusing,
because it looks like Listing 3.2 actually selects a node-set consisting of
all the child elements of the <code>entrees</code> node. If the
<code>entrees</code> node would also contain a text value, you would see that
this isn't true. We can however create an additional
<keyword>template</keyword> to handle the <code>dish</code> elements, as is
shown in Listing 3.4.
</analysis>
```

ANALYSIS Listing 4.10 shows a section of the preceding lesson. I tagged this text to show you what I mean by ad hoc structured data.

Note The tagging I used in Listing 4.10 is fictional. This book was created differently, but it could be done this way.

Documents like Listing 4.10 have no predetermined structure. At one point or another, the text can contain keywords, code, or other kinds of text requiring tagging. These tags can even be nested; for example, you might nest a keyword inside a piece of code. This type of document is quite different from a document with product information or one like the samples in the preceding section. Those documents have a more or less predetermined (and somewhat rigid) structure.

Besides being reusable entities, templates enable you to work with flexible data. Currently, most languages don't have this capacity, so it is one of the key benefits of XML and XSLT.

Creating and Using Templates

So far, you've seen some of the benefits of using templates, even though you haven't learned about some of the more powerful template features. The samples used to this point work with single matches and let the processor decide which template to invoke. If necessary, you can impose control over that process. The following sections look at various ways to gain more control.

More About Match Templates

You have already learned a great deal about match templates and have seen that they are powerful tools. This power mainly comes from the matching expressions you can use. The expressions used so far have been quite simple, matching only nodes of the same type. You also can create expressions that match nodes of different types. The easiest way to do so is to create two expressions and use them together in one expression.

NEW TERM You can add two expressions together by using the `|` character, also known as the *union* operator. Listing 4.11 shows a stylesheet using the union operator.

Note The union operator is not to be confused with the `OR` operator common in some languages. When you're matching templates, their function is more or less the same, but in XSLT, it can be used in other situations in which they are not similar.

LISTING 4.11 Stylesheet with an Expression Using the Union Operator

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

```

```
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="firstname|lastname">
10:     <xsl:value-of select="." />
11:   </xsl:template>
12: </xsl:stylesheet>
```

ANALYSIS The template on line 9 in Listing 4.11 operates on both the `firstname` and `lastname` elements. The expressions matching these elements are combined using the union operator. Listing 4.12 contains sample XML, and Listing 4.13 shows the result when Listing 4.11 is applied to Listing 4.12.

LISTING 4.12 Sample XML to Be Used with Listing 4.11

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person>
    <firstname>Joe</firstname>
    <lastname>Dishwasher</lastname>
  </person>
  <person>
    <firstname>Carol</firstname>
    <lastname>Waitress</lastname>
  </person>
</persons>
```

OUTPUT **LISTING 4.13** Result from Applying Listing 4.11 to Listing 4.12

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

    Joe
    Dishwasher


    Carol
    Waitress
```

ANALYSIS The result in Listing 4.13 shows that the template matches both elements and does exactly the same with each element—outputs its value. Why is this technique useful? Well, an XML document may contain data similar to another, but named differently and possibly structured differently. The sample document in Listing 4.14 contains the same data as Listing 4.4 but is structured differently.

LISTING 4.14 XML Document with Car Information

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
   <car>
      <model>Focus</model>
      <manufacturer>Ford</manufacturer>
      <year>2000</year>
   </car>
   <car>
      <model>Golf</model>
      <manufacturer>Volkswagen</manufacturer>
      <year>1999</year>
   </car>
   <car>
      <model>Camry</model>
      <manufacturer>Toyota</manufacturer>
      <year>1999</year>
   </car>
   <car>
      <model>Civic</model>
      <manufacturer>Honda</manufacturer>
      <year>2000</year>
   </car>
   <car>
      <model>Prizm</model>
      <manufacturer>Chevrolet</manufacturer>
      <year>2000</year>
   </car>
</cars>
```

ANALYSIS In Listing 4.14 the attributes of each `car` element in Listing 4.4 are replaced by elements. So each `car` element now has only child elements. These elements have the same names and values as the attributes in Listing 4.4. The challenge is now getting (more or less) the same output from both documents by using only one style-sheet. To do so, you need to create templates that match both attributes and elements for each element/attribute name. For instance, the match expression for the `model` element/attribute would be `model|@model`, so the `model` attribute is treated the same as the `model` element.

There is one problem. For Listing 4.14, this naming technique works fine because this example consists of elements only. When `xsl:apply-templates` is executed, the processor matches all elements it encounters. Attributes, however, are not matched, so applying these templates to Listing 4.4 would yield an empty result because all data is stored in attributes. You therefore need to specify that attributes should also be added to the node-set that the processor tries to match. To do so, you need to add a `select` attribute to the `xsl:apply-templates` element. This attribute's value should tell the processor which node-set to use when looking for a match. Like the `match` attribute of a template, the value of a `select` attribute is an XPath expression.

Be aware that a processor handles a template match expression and a select expression differently. As I discussed earlier, a template is matched and therefore depends on the node's place and sequence in the source document. A select expression, on the other hand, *selects* a node-set. The node sequence in this case depends on the select expression defining the node-set. You used select expressions previously with the `xsl:value-of` element. You may remember that these expressions are somewhat awkward when a select expression yields a node-set instead of a text value. The expression has to create the value of the selected node-set. That same node-set is what you select when you're using `xsl:apply-templates`. The idea is that you select the node-set that has to be matched, and then the processor takes over and starts matching each node in the node-set against the available templates. Listing 4.15 shows how you can transform Listings 4.4 and 4.14 with the same stylesheet.

LISTING 4.15 Stylesheet Operating on Listing 4.4 and Listing 4.14

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="car">
    <xsl:apply-templates select="*|@*" />
  </xsl:template>

  <xsl:template match="model|@model">
    Model:        <xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="manufacturer|@manufacturer">
    Manufacturer: <xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="year|@year">
    Year:         <xsl:value-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 4.15 produces a result similar to Listing 4.9 for both Listings 4.4 and 4.14. The templates on lines 13, 17, and 21 match either an element or an attribute (with the same name). They therefore perform the same action on the context node, making no difference between an element and an attribute. Line 10 in Listing 4.15 is very important because it tells the processor to select a node-set consisting of all the elements and attributes that are child nodes of the context node. In this case, all elements that are child elements of the `car` element and all attributes of the `car` element are part of the selected node-set. For Listing 4.4, this node-set consists solely of attributes, and for Listing 4.14, it consists solely of elements, but that makes no difference for the output.

Line 10 in Listing 4.15 contains a broad selection of *all* elements and attributes. You can easily limit this line to only certain elements and attributes. For instance, if you want a list of models only, you can change the value of the select expression to `model|@model`. This select expression selects only the child nodes that are either `model` elements or `model` attributes.

Using Named Templates

NEW TERM Templates don't necessarily have to be matched. They can be called directly as well, forcing the processor to execute the template. Using templates this way is much more like traditional programming in which you call a function or procedure that is to be executed. For you to be able to call a template explicitly, it must have a unique name within the stylesheet. This is why these types of templates are referred to as named templates. A *named template* is a template that can be called from another template, rather than being matched to a node by the processor.

You name a template by adding a `name` attribute with a unique identifier as its name. This identifier may contain a namespace, but if that's the case, that namespace must be declared either in the `stylesheet` element or as part of the template itself.

Note Namespaces are discussed thoroughly on Day 15, "Working with Namespaces," so don't worry if you're not familiar with them.

A big difference between match templates and named templates is that when a named template is called, the context node stays the same. With a matched template, the context node changes the moment `xsl:apply-templates` is used. Throughout the matching process, the context node can even change again.

Named templates are invoked using the `xsl:call-template` element. Which template is invoked is determined by the `name` attribute of `xsl:call-template`. The `name` attribute's value must be the name of an existing template within the stylesheet. If the template doesn't exist, an error occurs. Named and match templates do not interfere with one another. It is perfectly acceptable to have a template with a `name` attribute that is the same as a `match` attribute of another. Listing 4.16 shows an example with a named template.

LISTING 4.16 Stylesheet with Named Template and Match Templates

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="car">
10:     <xsl:call-template name="car" />
11:   </xsl:template>
12:
13:   <xsl:template name="car">
14:     Manufacturer: <xsl:value-of select="@manufacturer" />
15:     Model:        <xsl:value-of select="@model" />
16:     Year:         <xsl:value-of select="@year" />
17:   </xsl:template>
18: </xsl:stylesheet>
```

ANALYSIS **NEW TERM** Listing 4.16 contains a template matching the `car` element on line 9, and it also contains a template named `car` on line 13. The named template is called from the matched `car` template, at which time control is passed over to the named template without changing the context node. Hence, the `xsl:value-of` elements can use relative addressing and directly address the attributes of the `car` element without having to point to the location of those attributes. Even though the flow of control is entirely different for this stylesheet, it yields an output similar to Listing 4.9 when it is applied to Listing 4.4. The *flow of control* is the sequence in which commands or, in the case of XSLT, elements are executed.

Listing 4.16 is, of course, not very useful. The `car` element is processed by a different template that does exactly the same as match templates of earlier samples. Why are these templates useful? Well, mostly to break up complex templates into smaller units that are easier to work with.

NEW TERM Named templates give you complete control over the flow of control. This undermines the whole idea of the data-driven nature of XSLT. However, in some cases, you might need to impose such control. One of those instances occurs when you need *recursion,* which happens when a function or, in the case of XSLT, a template calls itself. This is often the case when operations need to be executed several times based on some condition.

Note On Day 17, "Using Recursion," recursion will be discussed further.

Combining Named Templates and Match Templates

A match template and a named template can be one and the same. A template must have a match expression or a name, but can also have both. In that case, the template can be either matched or called, whichever is appropriate at that point.

Determining Which Template Is Used

In the preceding sections, you learned that you can explicitly tell the processor which template to process. You also learned that you can impose some control over which templates can be matched. But what happens when a node is matched by more than one template? When more than one template applies, their precedence rules determine which template is actually used. You can influence the precedence using priorities. You also can make different templates for different uses. You determine which template is used in which instance. In the following sections, these topics will be discussed.

Different Templates for Different Cases

You will find that in some cases you may have to go through a document more than once, each time creating different output. In these situations, you need several different templates matching the same nodes. Each template takes a different action. The problem is that the processor cannot determine which template to use in the separate instances. To ensure that the processor invokes the correct template, you can add a `mode` attribute to each template. The value of the `mode` attribute can be any name, but it needs to be different for each template matching the same node or node-set, and the same for each template to be invoked in a mode. When you `xsl:apply-templates`, you specify which mode to use. The processor then matches using only those templates belonging to that particular mode. Using this approach, you can ensure that each time you go through the document different tasks will be performed, as shown in Listing 4.17.

LISTING 4.17 Stylesheet Employing Modes

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
```

```
5:    <xsl:template match="/">
6:      <xsl:apply-templates mode="index" />
7:      <xsl:apply-templates mode="info" />
8:    </xsl:template>
9:
10:   <xsl:template match="car" mode="index">
11:     <xsl:value-of select="@model" />
12:   </xsl:template>
13:
14:   <xsl:template match="car" mode="info">
15:    <xsl:value-of select="@model" />
16:    Manufacturer: <xsl:value-of select="@manufacturer" />
17:     Year:         <xsl:value-of select="@year" />
18:   </xsl:template>
19: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 4.17 contains two different modes, invoked one after another. Line 6 invokes templates using the `index` mode, and line 7 invokes templates using the `info` mode. When each mode is invoked, only the templates that are part of that particular mode are matched. This way, the document is processed twice, each time with a different mode. Line 10 defines a template that uses the `index` mode. This template is invoked only by line 6. The template on line 14 uses the `info` mode and is invoked only by line 7. Listing 4.18 shows the result of applying Listing 4.17 to Listing 4.4.

OUTPUT **LISTING 4.18** Result from Applying Listing 4.17 to Listing 4.4

```
<?xml version="1.0" encoding="utf-8"?>
  Focus
  Golf
  Camry
  Civic
  Prizm

  Focus
    Manufacturer: Ford
    Year:         2000
  Golf
    Manufacturer: Volkswagen
    Year:         1999
  Camry
    Manufacturer: Toyota
    Year:         1999
  Civic
    Manufacturer: Honda
    Year:         2000
  Prizm
    Manufacturer: Chevrolet
    Year:         2000
```

ANALYSIS In Listing 4.18, the document from Listing 4.4 is processed twice. The first time only the car models are listed. On the second run, each car is listed by model, and the manufacturer and year are added.

Using modes is useful when you need to create a document that starts with a table of contents and also contains the entire text or all the data. For example, you see Web pages where the start of the page has a table of contents that enables you to go directly to the section of the page that you're interested in. Modes are not necessarily restricted to the entire document. You also can use modes on just a section of a document tree. You can even change modes from one template to another. This way, you can create a complex flow of control, creating complex output.

Modes can also help you when you have different formatting for differently structured XML sources. In that case, you can create a stylesheet that detects which structure is used and then use the appropriate mode to process the document. This way, nodes that have to be processed differently for that structure are treated differently. Be aware that only templates belonging to a certain mode are matched when that mode is used. Templates that do not have a `mode` attribute are treated as a separate mode. This means that templates that have no mode are matched only when no mode is used.

Template Priorities

Keeping templates apart using modes is very handy, but when templates with the same mode match, you need to know which template will actually be invoked. The processor selects the actual template that is invoked by going through a series of selection steps as follows:

1. The processor selects all the templates that have a `match` attribute.
2. From those templates, the processor selects all templates that have the same mode as the active mode.
3. The processor selects those templates whose match expression matches the current node.
4. If more than one template has a match expression that matches the current node, the template that has the highest numerical priority is selected.
5. If more than one template is still left, the processor can report an error or use the last template that occurs in the stylesheet.

The priority in step 4 is a positive or negative number. A template's priority can be specified explicitly. If it is not, it is calculated based on the match expression. Typically, match expressions that are more precise have a higher numerical priority. For instance, an expression using a wildcard has a lower priority than a match expression specifically matching a node. Multiple expressions added to one another using the union operator (`|`) are treated as separate expressions. For each expression, the priority is calculated separately. Table 4.1 shows the default priorities.

TABLE 4.1 Default Template Priorities

Expression	*Priority*
Specific child element (for instance, `car` or `child::car`)	0.0
Specific attribute (for instance, `@model` or `attribute::model`)	0.0
Processing instructions (for instance, `processing-instruction('mypi')`)	0.0
Wildcard child element in a specific namespace (for instance, `cars:*` or `child::cars:*`)	-0.25
Wildcard attribute in a specific namespace (for instance, `@car:*` or `attribute::car:*`)	-0.25
Wildcard child element (for instance, `*` or `child::*`)	-0.5
Wildcard attribute (for instance, `@*` or `attribute::*`)	-0.5
Other node tests (for instance, `text()` or `node()`)	-0.5
Node tests outside the current axis	0.5
Expressions with predicates	0.5

Table 4.1 looks more complex than it is. Basically, it shows that the less specific an expression is, the lower its priority. Expressions with node tests that are outside the current axis or with predicates have a higher priority because the expression more specifically states which node is to be matched. Note that this also means that an expression such as `//node()` has a higher priority than an expression that specifies a child element. This prioritization is somewhat strange because the former expression actually matches any node within the document tree. This example goes against the idea that the more precise an expression is, the higher the priority will be. Listing 4.19 gives you an idea of the default priorities.

LISTING 4.19 Stylesheet Showing Default Priorities

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <xsl:apply-templates />
11:   </xsl:template>
12:
13:   <xsl:template match="car">
14:     Model: <xsl:value-of select="@model" />
15:   </xsl:template>
16:
17:   <xsl:template match="*">
18:     xyz
19:   </xsl:template>
20: </xsl:stylesheet>
```

ANALYSIS The template on line 17 uses a wildcard to match elements. Theoretically, it will match any element with any name. The templates on lines 9 and 13 match specific elements and should have precedence over the wildcard template. If you apply Listing 4.19 to Listing 4.4, the default priorities ensure that the template matching the `car` element is invoked for the `car` elements in Listing 4.4. If that weren't the case, the template on line 17 using a wildcard would be invoked instead. Listing 4.20 shows the result of applying Listing 4.19 to Listing 4.4.

OUTPUT **LISTING 4.20** Result from Applying Listing 4.19 to Listing 4.4

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

    Model: Focus

    Model: Golf

    Model: Camry

    Model: Civic

    Model: Prizm
```

ANALYSIS In Listing 4.20, the template matching the `car` element is invoked for each `car` element in Listing 4.4. The template with the wildcard match isn't invoked anywhere. If the listing didn't have any priorities, the template with the wildcard would be the last template to occur in the stylesheet and hence would be the template invoked. Because of the default priority, the template matching the `car` element is invoked as it should be.

As your stylesheets become larger, default priorities will be more and more important. You are likely to encounter situations in which the invoked template is different from the template you thought would be invoked. In those cases, an XSLT debugger is useful because it shows which template is invoked. Getting the processor to invoke the right templates at that point may prove tricky, but don't let the difficulty slow you down. When you gain more experience using XSLT, you will learn to avoid these situations most of the time and remedy them quickly when they do occur.

Adding Your Own Priorities

The default priorities calculated by the processor are always between -0.5 and 0.5. This leaves you free to define priorities that override the default priorities. If you define a priority, it needs to be a whole number, such as 1, 5, or 13. Specifying a priority with a value lower than -0.5 would make no sense because it would never be invoked; the default priority would always be higher.

You can specify your own priority for a template by adding a `priority` attribute to the `xsl:template` element. This attribute needs to have a value that is a positive whole number. Listing 4.21 shows a stylesheet using priorities.

LISTING 4.21 Stylesheet Using Priorities

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <xsl:apply-templates />
11:   </xsl:template>
12:
13:   <xsl:template match="car">
14:     Model: <xsl:value-of select="@model" />
15:   </xsl:template>
16:
```

LISTING 4.21 Continued

```
17:   <xsl:template match="*" priority="1">
18:     xyz
19:   </xsl:template>
20: </xsl:stylesheet>
```

ANALYSIS Listing 4.21 is similar to Listing 4.19 except that line 17 defines a template with a `priority` attribute. The value of the `priority` attribute is `1`, so it will override any default priorities. So, even if the match expression of the other templates is more specific, the wildcard template on line 17 will be invoked. Listing 4.22 shows the result of applying Listing 4.21 to Listing 4.4.

OUTPUT **LISTING 4.22** Result from Applying Listing 4.21 to Listing 4.4

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

ANALYSIS The output in Listing 4.22 is short because the `cars` element in Listing 4.4 is matched by the template on line 17 in Listing 4.21 using the wildcard match instead of the template matching the `cars` element specifically on line 9. Because this template doesn't contain the `xsl:apply-templates` element, no further processing occurs after this template is invoked and the value `xyz` is written to the output.

From the example in Listing 4.22, you may think that defined priorities are not very useful. This is because of the simplicity of the XML documents and the stylesheets used so far. In more complex documents and stylesheets, priorities are more valuable—specifically in cases in which the match expressions are general and more than one template matches. If you have a specific matching template, you don't have a problem. However, if all matching templates are quite general, you don't know for certain which template matches a certain node. One solution would be to write expressions that are less general. However, doing so would mean writing more code for the same task because you would probably have to create more templates for that task. Therefore, priorities become useful when you have a very wide base of code reuse.

Summary

Today you learned that templates are structures that can be filled with data. Throughout a stylesheet, templates can be used as blocks of reusable code. Match expressions determine for which elements and attributes a template is used. These match expressions can be very complex, and one expression can even be the union of multiple expressions.

If more than one template matches the current node, the processor takes a series of steps to determine which template to invoke. These steps involve selecting the templates of the current mode that match the node and using the defined or calculated priority. The general rule for calculated priorities is that the more precisely an expression matches the current node, the higher the priority. Because of the way these rules work, you may have a surprise or two.

Today you also learned that you can invoke templates by calling them rather than have the processor match a template. This way, you can break up the complex code of a matched template because the context node stays the same when you call another template. A template must have a match expression or a name to be called with, but may have both.

Tomorrow you will learn how to create more complex output that can contain text with elements and attributes. This way, you can create XML documents from other XML documents and, in the process, restructure the XML.

Q&A

Q Can I create a stylesheet without templates?

A If you want something to happen in a controlled fashion, you need at least a template matching the root of the XML document. If you can create all the output from the `xsl:value-of` elements, that is up to you. You will learn more details about such elements in the following lessons, but because using templates is the preferred method and key to XSLT, templates have been discussed thoroughly first.

Q Can I use modes to create multiple documents?

A Just using modes is not enough to create multiple documents. You can create output with a divider and split the resulting document into more documents by using another program.

Q Can called templates have a mode?

A No. Named templates need to have a unique name within the document. Hence, differentiating between them with modes makes no sense.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: A template can have a `match` attribute and a `name` attribute.
2. True or False: A template using a wildcard expression in the current context has a higher priority than an expression targeting a specific node in the current context.
3. If `xsl:apply-templates` is used with a mode, will templates that have no mode be matched?
4. Why do you use a `select` attribute with `xsl:apply-templates`?
5. What is the difference between selecting nodes and matching nodes?

Exercises

1. Create a stylesheet for Listing 4.4 to display a list of models, then a list of manufacturers, and finally a list of years.
2. Create a stylesheet with the same output as Exercise 1, but for Listing 4.14. You also may change the stylesheet from the preceding exercise to handle both listings.

DAY 5

Inserting Text and Elements

Yesterday you learned all about using templates, that they are key to Extensible Stylesheet Language Transformations (XSLT), and that they can either be explicitly called or matched by the processor. When templates are being matched, multiple templates may match the current node. You have learned how to control which template is actually matched when this situation occurs, enabling you to control the output that is actually generated.

In today's lesson, you will learn more details about inserting text and elements into the resulting output. You will learn how to do the following:

- Insert text with special characters
- Insert elements and attributes into the XML result
- Insert elements and attributes with a generated name
- Copy elements from the source XML into the result

Inserting Text

As you learned in previous lessons, inserting text is easy. In a template, you specify the text you want, and it is then inserted each time the template is processed. When you want to insert the value from an element or attribute in the source XML document into the result, you can use the `xsl:value-of` element to specify which value you want to insert.

You may be wondering why this section is devoted entirely to inserting text if the procedure is so simple. Well, I included this discussion because the text inserted and the values in elements and attributes have been plain in the previous samples. I specifically avoided special characters that might cause problems in those samples. Because you will most certainly encounter situations in which you need to work with characters such as é, ë, ê, and so on, a closer look is in order.

Text with Special Characters

NEW TERM Unless you specify otherwise, a stylesheet will generate XML. In XML, and thus in XSLT, some characters need to be treated differently. They are called *special characters*. Generally, special characters are a problem when a document is processed—either because of the encoding used or because these characters have a special function within the document (for example, tag delimiters).

NEW TERM To use special characters in a value (in text), you need to use output escaping. Characters are said to be *output escaped* when they are replaced by a series of characters that represent them. You use them if you want to insert those characters without their performing the function they have within the document, or when those characters are not supported by the encoding used and hence need to be represented in some other way.

Output escaping is common in HTML documents because these documents are based on ASCII encoding, which supports only 256 characters. Many characters used in different languages are not among those 256 characters, so to represent them, you need to use some kind of alternative encoding. Because XML can be based on many encoding schemes, including Unicode, XML by default requires only minimal output escaping. In Unicode, most common international characters can be inserted as is, so you don't need to replace characters such as é, ë, and ê with other characters to represent them. If your policy is to always output escape these kinds of characters, you are certainly allowed to do so.

NEW TERM In XML, output escaping works on the basis of entities. An *entity* is a name that represents a character or series of characters. It can be used as a replacement for a special character or as a shortcut to insert a value (for example, a copyright notice).

By default, XML defines only five entities, which replace characters within XML that have a special function. Table 5.1 shows the characters for which entities have been defined in XML.

TABLE 5.1 Default Entities Defined in XML

Character	*Entity*
&	`&amp;`
<	`&lt;`
>	`&gt;`
“	`&quot;`
‘	`&apos;`

Of the entities in Table 5.1, only the first two always must be used. The `&amp;` entity always denotes the start of an entity that is being inserted, including the one replacing it. The `&lt;` entity always denotes the start of an XML tag, so it can never be used in regular text. You must use the other three entities only when the processor may interpret a character's presence as performing the function it has in XML rather than being part of the text. If, at that point, you want to insert that character in the text, you need to insert the entity. When the actual character's presence cannot be misinterpreted, you can insert that character as is, as shown in Listing 5.1.

LISTING 5.1 Sample with Different Forms of Output Escaping

```
<?xml version="1.0" encoding="UTF-8"?>
<escaping>
  <amp>&amp;</amp>
  <lt>&lt;</lt>
  <quot>&quot;</quot>
  <quot attr="&quot;">"</quot>
  <apos attr='&apos;'>&apos;</apos>
  <apos>'</apos>
  <gt>&gt;</gt>
  <gt attr="&gt;">></gt>
  <other>&#100;</other>
  <other>&#x00E7;</other>

</escaping>
```

Note

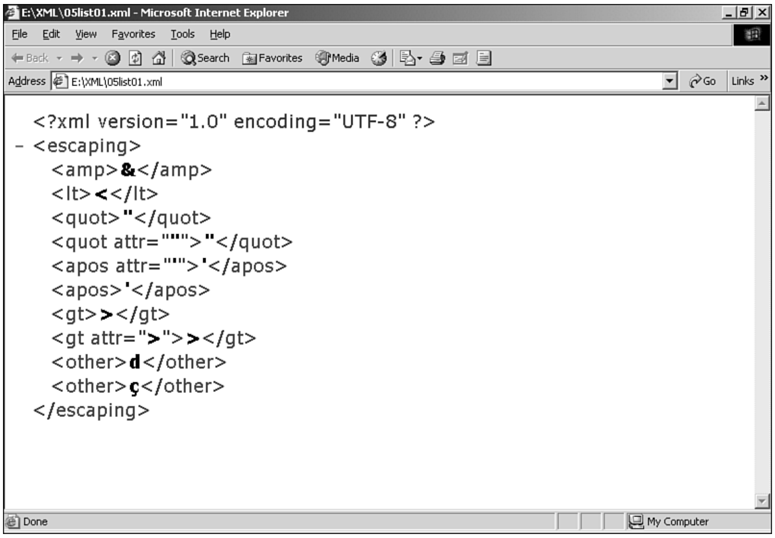
You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS Listing 5.1 contains quite a mix of escaped characters. Where possible, the characters are inserted as is. For the values of the elements on lines 3–4, using entities instead of the actual characters is mandatory. Not doing so would result in an error. For the quote character on line 6, you can use either the quote character within an element value or use the corresponding entity. Because the element's attribute on line 6 uses the quote character to delimit the value, the quote entity is needed to insert the actual character in the value. For the apostrophe character, this use is similar, as shown on lines 7–8. The > character also doesn't need to be output escaped, except when it is within an attribute value. Otherwise, it could be mistaken for the element tag's ending. The characters on lines 11–13 don't need to be escaped, but you can choose to do so.

If you look at the code for Listing 5.1 in an XML-enabled viewer, such as Internet Explorer 5.0 or higher, you can see that the output-escaped characters actually appear as the characters themselves, as shown in Figure 5.1.

FIGURE 5.1

Screenshot of Listing 5.1 viewed in Internet Explorer 5.


E:\XML\05list01.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\XML\05list01.xml
Go Links
Done
My Computer

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <escaping>
    <amp>&</amp>
    <lt><</lt>
    <quot>"</quot>
    <quot attr=""">"</quot>
    <apos attr="'">'</apos>
    <apos>'</apos>
    <gt>></gt>
    <gt attr=">">></gt>
    <other>d</other>
    <other>ç</other>
  </escaping>
```

HTML Entities

If you're familiar with HTML, you may have noticed that entities such as `&nbsp;` (non-breaking space) are no longer defined. Actually, inserting such an entity by itself results in an error. For `&nbsp;` you should use `&#160;` instead. The five built-in entities in XML will be properly processed by all parsers on all platforms. You can extend entities either by using numerical codes or by creating your own with a DTD.

Note

You can find a complete list of the entities defined in HTML 4.0 at `http://www.w3.org/TR/REC-html40/sgml/entities.html`. For each entity listed, the equivalent number code also is given. These number codes, which correspond to the number codes in Unicode, can be used in XML, as can *all* other Unicode number codes.

Another option is to create a Document Type Definition (DTD) that defines these entities and use that DTD when you're creating documents and stylesheets in which you want to use these entities.

Note

Defining entities with DTDs is beyond the scope of this book. Using DTDs is discussed further on Day 15, "Working with Namespaces."

Special Characters in XSLT

The bottom line from the preceding discussion is that XML can contain special characters, but some characters have to be output escaped because of their function in XML. The question that now arises is "How does XSLT deal with these characters?" To answer this question, look at Listing 5.2.

LISTING 5.2 Stylesheet with Special Characters

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="apos">
    &quot;<xsl:value-of select="." />&quot;
  </xsl:template>

  <xsl:template match="gt">
    >&quot;<xsl:value-of select="." />"&lt;
  </xsl:template>

  <xsl:template match="amp|lt|quot|other">
    &apos;<xsl:value-of select="." />'
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 5.2 contains several templates that could be used with Listing 5.1. In the templates, some characters are output escaped, but others aren't just to show you that you can mix up the characters when it doesn't matter whether you use output escaping. In the template starting on line 13, one of the quote characters is output escaped; the other is not. Also, the > that starts the template's value is not output escaped. It can be used this way because it is in a position where the parser does not expect a > character to perform a function (ending a tag). The < character ending the value needs to be output escaped, however, if you don't want the parser to mistake it for the start of a tag.

Listing 5.3 shows the result of applying Listing 5.2 to Listing 5.1.

OUTPUT **LISTING 5.3** Result from Applying Listing 5.2 to Listing 5.1

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

    '&amp;'


    '&lt;'


    '"'


    '"'


    "'"


    "'"


    &gt;"&gt;"&lt;


    &gt;"&gt;"&lt;


    'd'


    'ç'

```

ANALYSIS In Listing 5.3, most output-escaped characters are replaced by their actual characters. The reason for this conversion has already been hinted at in Figure 5.1. When the parser loads the XML, the characters it can handle without their being output escaped are immediately converted to the characters themselves. In effect, after the XML is loaded, you cannot tell whether the characters were output escaped. This is why in XSLT you cannot determine whether a character in the source XML was output escaped.

Note If you apply Listing 5.2 to Listing 5.1 from the command prompt or console, the output looks a little different because the command prompt doesn't have the means to display some of the characters. If you save the result to a file and view that file in a text editor that supports these characters, the characters are displayed as they should be.

When the output is created, the processor outputs all the characters conforming to the output encoding used (in this case, UTF-8), regardless of the output escaping used in the source XML or stylesheet. If the processor determines that a character does need to be output escaped, it outputs the character by itself, also regardless of the way the character appeared in the input. Refer to line 10 of Listing 5.1 and the output generated from that line with the stylesheet, which appears on line 24 of Listing 5.3. As you can see, the processor automatically converts the > character to its entity.

As long as your XML source and stylesheets are output escaped properly, you don't need to worry about what the resulting output will look like from an output-escaping point of view. When you're creating XML output, the processor ensures that the output is encoded and output escaped as it should be. When you create other forms of output, you may have to be more careful, however.

Note On Day 7, "Controlling the Output," different output types will be discussed further.

Inserting Elements and Attributes

I have been saying that the samples in this book produce XML output. Strictly speaking, this is true because all the output is preceded by an XML prolog, and all characters conform to XML encoding and output escaping. You may have noticed that none of the results contain tags; they contain only text. The output created therefore is not well-formed XML because well-formed XML documents need to have at least a root element.

Values in the document must exist inside that root element; otherwise, the document is not valid. So, to create valid XML documents, you need to be able to insert elements.

Inserting Elements

Like inserting text, inserting elements is straightforward. You can insert elements anywhere within a template, as long as you use well formed XML (if it isn't, the stylesheet itself will not be well formed XML). If you want to insert elements with a namespace, that namespace needs to be declared within the stylesheet.

Note On Day 15, "Working with Namespaces," inserting elements with a namespace will be discussed thoroughly.

Inserting elements in the output is required if you want the output to be XML or HTML. If you restructure your source document or if you format the source document for display on the Web, this is the case. When you're transforming elements to HTML, for instance, you may want to insert hyperlinks, text in boldface, and so on.

The best way to show how easily you can insert elements is by example. The first thing you need is an XML document to be transformed. The document used in Listing 5.4 comes from yesterday's lesson.

LISTING 5.4 XML Sample Document with Car Information

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

Now you need a stylesheet to transform Listing 5.4 and add some elements. Listing 5.5 shows such a stylesheet.

LISTING 5.5 Stylesheet That Inserts Elements into the Output

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
```

```
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <table>
11:       <xsl:apply-templates />
12:     </table>
13:   </xsl:template>
14:
15:   <xsl:template match="car">
16:     <tr>
17:       <xsl:apply-templates select="@*" />
18:     </tr>
19:   </xsl:template>
20:
21:   <xsl:template match="@*">
22:     <td><xsl:value-of select="." /></td>
23:   </xsl:template>
24: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 5.5 creates an HTML table containing all the information on the cars stored in Listing 5.4. For each `car` element, a row is created in the table. The `cars` template starting on line 9 creates the start tag of the `table` element and then uses the `xsl:apply-templates` element to invoke other templates for each child element of the `cars` element. When that has been done, the end tag of the `table` element is inserted on line 12. The `car` template starting on line 15 is invoked several times, for each `car` element in Listing 5.4. This template inserts a table row starting tag on line 16, and the closing tag on line 18. In between, it invokes the processor for all its attributes. For each attribute, a table cell is created by the template starting on line 21, which also inserts the value of the attribute. When you apply Listing 5.5 to Listing 5.4, you get the result shown in Listing 5.6.

OUTPUT **LISTING 5.6** Result from Applying Listing 5.5 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><table>
  <tr><td>Focus</td><td>Ford</td><td>2000</td></tr>
  <tr><td>Golf</td><td>Volkswagen</td><td>1999</td></tr>
  <tr><td>Camry</td><td>Toyota</td><td>1999</td></tr>
  <tr><td>Civic</td><td>Honda</td><td>2000</td></tr>
  <tr><td>Prizm</td><td>Chevrolet</td><td>2000</td></tr>
</table>
```

If you open Listing 5.6 as an HTML file in a browser, you get a neat table with cars.

Note

Technically, Listing 5.6 is XML. However, because the entire document conforms to the rules of HTML, you can save it as an HTML file and view it in a Web browser.

Analysis The stylesheet in Listing 5.5 could also have created non-HTML tags. In that case, Listing 5.6 would look different, but it would have been well-formed XML as well. Nothing restricts you to using only HTML tags. I chose to generate HTML tags to give you a practical example that might occur in the real world. I created the HTML tags in such a way that the result is well-formed XML as well. Therefore, you can load the result into a viewer or parser without generating an error. Some HTML tags are not well-formed XML, such as `<br>` and `<input>`. If you want to insert such tags, you need to make them well formed—for instance, `<br></br>` or `<input />`. If you don't, the stylesheet itself is not well-formed XML, so an error occurs when you try to use it. When these tags are sent to the output as well-formed XML, they may not always appear correctly in a Web browser.

Note

You'll learn how to create correct HTML output on Day 7, "Controlling the Output."

Creating a Different XML Structure

Listing 5.4 contains `car` elements, with the values stored in attributes. Yesterday you saw a listing with the same information, but stored in child elements. You created a stylesheet to generate the same output for both structures. Another option is to restructure one of the documents so that it uses the same structure as the other. The stylesheet in Listing 5.7 does exactly that.

Listing 5.7 Stylesheet to Restructure Listing 5.4

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <cars>
```

```
11:       <xsl:apply-templates />
12:     </cars>
13:   </xsl:template>
14:
15:   <xsl:template match="car">
16:     <car>
17:       <xsl:apply-templates select="@*" />
18:     </car>
19:   </xsl:template>
20:
21:   <xsl:template match="@model">
22:     <model><xsl:value-of select="." /></model>
23:   </xsl:template>
24:
25:   <xsl:template match="@manufacturer">
26:     <manufacturer><xsl:value-of select="." /></manufacturer>
27:   </xsl:template>
28:
29:   <xsl:template match="@year">
30:     <year><xsl:value-of select="." /></year>
31:   </xsl:template>
32: </xsl:stylesheet>
```

ANALYSIS Listing 5.7 isn't very different from Listing 5.5. The `cars` template on line 9 inserts a `cars` element instead of a `table` element, and the `car` template on line 15 inserts a `car` element instead of a `tr` element. The major difference between Listings 5.7 and 5.5 is the way the `car` element's attributes are handled. In Listing 5.5, all attributes are handled by the same template, which inserts a `td` element and the value of the attribute being processed. In Listing 5.7, each attribute is handled by a different template, to see to it that the newly created elements have the same name as the attribute in the source XML. When you apply Listing 5.7 to Listing 5.4, you get the results shown in Listing 5.8.

OUTPUT **LISTING 5.8** Result from Applying Listing 5.7 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><cars>
  <car><model>Focus</model><manufacturer>Ford</manufacturer>
➥<year>2000</year></car>
  <car><model>Golf</model><manufacturer>Volkswagen</manufacturer>
➥<year>1999</year></car>
  <car><model>Camry</model><manufacturer>Toyota</manufacturer>
➥<year>1999</year></car>
  <car><model>Civic</model><manufacturer>Honda</manufacturer>
➥<year>2000</year></car>
  <car><model>Prizm</model><manufacturer>Chevrolet</manufacturer>
➥<year>2000</year></car>
</cars>
```

Analysis In Listing 5.8, the result contains all attributes from Listing 5.4 as elements. Although Listing 5.8 isn't as neatly formatted as Listing 4.14 in yesterday's lesson, they are syntactically and semantically identical. When the documents are loaded into a parser, the resulting XML document tree is the same for all intents and purposes.

Inserting Elements with a Generated Name

Earlier you learned how to insert elements literally into a stylesheet. This approach makes it easy to insert elements. You also can use another method to insert elements. This method uses `xsl:element` to insert elements. Using this XSLT element, you could write

```
<xsl:element name="model">Focus</xsl:element>
```

instead of

```
<model>Focus</model>
```

Using this method, as shown in Listing 5.9, you could also achieve the result in Listing 5.6.

Listing 5.9 Stylesheet Using `xsl:element`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="cars">
    <xsl:element name="table">
      <xsl:apply-templates />
    </xsl:element>
  </xsl:template>

  <xsl:template match="car">
    <xsl:element name="tr">
      <xsl:apply-templates select="@*" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="@*">
    <xsl:element name="td">
      <xsl:value-of select="." />
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listings 5.5 and 5.9 are nearly identical. In Listing 5.5, the elements are literally inserted; in this listing, the `xsl:element` element is used on lines 10, 16, and 22 to insert the elements. The result is identical. Writing the elements into the stylesheet literally is a shortcut for the method used in Listing 5.9.

You might be wondering why you would use `xsl:element` instead of inserting the elements literally. In the preceding situation, using `xsl:element` is indeed unnecessary. The code is easier to create and easier to read when you just insert the elements literally. The situation becomes different when you want to create elements in which the element names can be determined only at runtime. Literal elements have to be determined at design time rather than runtime, so inserting elements literally is not a possibility. You might be able to create templates with very complex match expressions to have more diversity in the elements you create, but the more complex the source XML processed becomes, the harder it becomes to deal with all the possibilities.

To get around the drawback of not being able to create elements dynamically at runtime, you can use `xsl:element` because the value of the `name` attribute of `xsl:element` can be determined at runtime using XPath expressions. This means that you can create elements that have a name given as a value in the source XML. Listing 5.10 shows how this technique works.

LISTING 5.10 Stylesheet Creating Dynamic Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="cars">
    <caryears>
      <xsl:apply-templates />
    </caryears>
  </xsl:template>

  <xsl:template match="car">
    <xsl:element name="{@model}">
      <xsl:value-of select="@year" />
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Applying Listing 5.10 to Listing 5.4 yields the result in Listing 5.11.

OUTPUT **LISTING 5.11** Result from Applying Listing 5.10 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><caryears>
  <Focus>2000</Focus>
  <Golf>1999</Golf>
  <Camry>1999</Camry>
  <Civic>2000</Civic>
  <Prizm>2000</Prizm>
</caryears>
```

ANALYSIS Listing 5.11 shows a list of car model years. You create the elements with the year value by using the `xsl:element` tag, as shown on line 16 in Listing 5.10. Line 17 inserts the corresponding year. The result is that each element's name corresponds to the value of each car's `model` attribute in Listing 5.4. Instead of using a literal value for the `xsl:element` tag's `name` attribute, you use an expression. You can see that it is an expression because of the curly braces surrounding the expression `@model`. Instead of creating elements like `<{@model}>`, the curly braces and expression are replaced with the value of the expression by the processor.

The expression on line 16 in Listing 5.10 is simple. You can make it more complex, getting a value from an entirely different part of the source XML. Also, you can mix expressions and literal text in the value of the `name` attribute, as shown in Listing 5.12.

LISTING 5.12 Creating an Element from Literal and Dynamic Values

```
1:    <xsl:template match="car">
2:      <xsl:element name="{@manufacturer}-{@model}">
3:        <xsl:value-of select="@year" />
4:      </xsl:element>
5:    </xsl:template>
```

ANALYSIS You can use the template in Listing 5.12 instead of the `car` template starting on line 15 in Listing 5.10. The value of the `name` attribute on line 2 in this listing contains two expressions surrounded by curly braces, with some literal text in between. If you apply the complete stylesheet of Listing 5.12 to Listing 5.4, the result is similar to Listing 5.13.

OUTPUT **LISTING 5.13** Result from Applying Listing 5.12 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><caryears>
  <Ford-Focus>2000</Ford-Focus>
  <Volkswagen-Golf>1999</Volkswagen-Golf>
  <Toyota-Camry>1999</Toyota-Camry>
  <Honda-Civic>2000</Honda-Civic>
  <Chevrolet-Prizm>2000</Chevrolet-Prizm>
</caryears>
```

ANALYSIS In Listing 5.13, the text and expressions comprising the value of the `name` attribute on line 2 of Listing 5.12 create elements with names that consist of the values of the `manufacturer` and `model` attributes, with a hyphen (`-`) in between. As long as the resulting element name is valid in XML, any combination of expressions and literal values is fine.

Using the Name of an Element or Attribute

In Listing 5.7, each attribute had to be handled by a different template to restructure the document. If you could use the attribute's name to create an element, that approach would not have been necessary. The XPath function `name()` comes into play here. This function gives the name of the context node. Listing 5.14 shows it in action.

LISTING 5.14 Stylesheet Using the `name()` Function

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

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

  <xsl:template match="car">
    <car>
      <xsl:apply-templates select="@*" />
    </car>
  </xsl:template>

  <xsl:template match="@*">
    <xsl:element name="{name()}"><xsl:value-of select="." /></xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS When Listing 5.14 is applied to Listing 5.4, the result is the same as Listing 5.8. This stylesheet, however, requires a lot less code than the one in Listing 5.7. In Listing 5.14, all the attributes are matched by the template that starts on line 21. Using the `name()` function, this template creates a new element with the name of the context node on line 22. In each case, this name is the name of the attribute. The value of the attribute becomes the value of the newly created element. The templates in lines 5, 9, and 15 all perform functions discussed earlier.

Using the method from Listing 5.14, you can actually create a stylesheet that works on every XML document that has attributes and convert that document so that it uses only elements. If you apply that same stylesheet to an XML document that has no attributes, the output is the same as the source document. The code to convert attributes is remarkably simple; it's shown in Listing 5.15.

LISTING 5.15 Stylesheet Converting Attributes to Elements

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="*">
6:      <xsl:element name="{name()}">
7:        <xsl:apply-templates select="@*" />
8:        <xsl:apply-templates />
9:      </xsl:element>
10:   </xsl:template>
11:
12:   <xsl:template match="@*">
13:     <xsl:element name="{name()}">
14:       <xsl:value-of select="." />
15:     </xsl:element>
16:   </xsl:template>
17:
18:   <xsl:template match="text()">
19:     <xsl:value-of select="." />
20:   </xsl:template>
21: </xsl:stylesheet>
```

ANALYSIS The template that starts on line 5 in Listing 5.15 will match any element in the document because it uses a wildcard. When it is invoked, it creates an element with the same name as the element for which the template is invoked, the context node. Then `xsl:apply-templates` invokes the processor for each attribute of that element. The template that starts on line 12 will match all these attributes because the attribute wildcard expression matches any attribute. This template creates an element with the name of the attribute being processed and gives it the value of that same attribute. After all the attributes are processed, control reverts to the template on line 5, which then invokes the processor for all child elements. These elements are either matched by the template itself or by the template starting on line 18, which just writes the value to the output.

As you can see, Listing 5.15 does not contain a template that matches the root element of the source document. Because of the built-in template rule, such a template isn't necessary.

Inserting Attributes

Inserting attributes isn't much different from inserting elements. If you insert elements literally, you can add attributes literally as well. Listing 5.16 shows how to do so.

LISTING 5.16 Partial Stylesheet Inserting Literal Attributes

```
1:    <xsl:template match="cars">
2:      <table border="1" width="500">
3:        <xsl:apply-templates />
4:      </table>
5:    </xsl:template>
6:
7:    <xsl:template match="car">
8:      <tr bgcolor="#dddddd">
9:        <xsl:apply-templates select="@*" />
10:     </tr>
11:   </xsl:template>
```

ANALYSIS The templates in Listing 5.16 are replacements for the `cars` and `car` templates on lines 9 and 15 in Listing 5.5. On line 2 of Listing 5.16, two attributes are added to the `table` element that line 1's template inserts. On line 8, the `tr` element now has a `bgcolor` attribute. The result is that each time these templates are invoked, the attributes are inserted along with the elements, as you can see in the result in Listing 5.17.

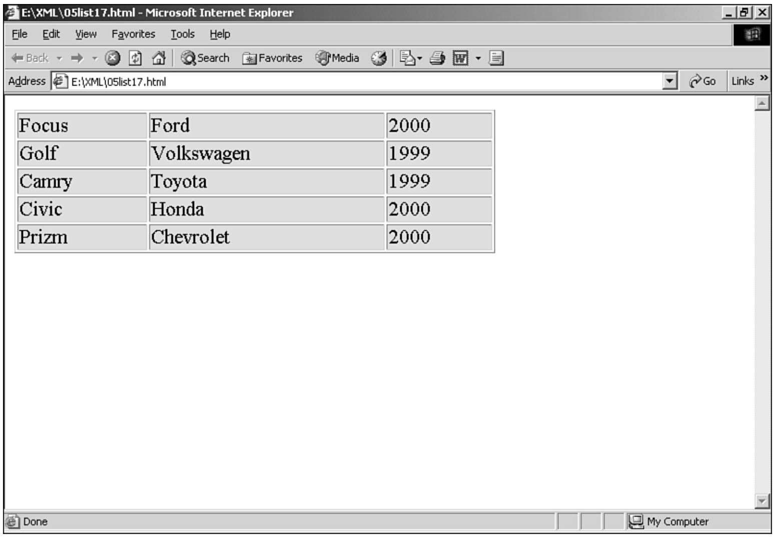
OUTPUT **LISTING 5.17** Result from Applying Listing 5.16 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><table border="1" width="500">
  <tr bgcolor="#dddddd"><td>Focus</td><td>Ford</td><td>2000</td></tr>
  <tr bgcolor="#dddddd"><td>Golf</td><td>Volkswagen</td><td>1999</td></tr>
  <tr bgcolor="#dddddd"><td>Camry</td><td>Toyota</td><td>1999</td></tr>
  <tr bgcolor="#dddddd"><td>Civic</td><td>Honda</td><td>2000</td></tr>
  <tr bgcolor="#dddddd"><td>Prizm</td><td>Chevrolet</td><td>2000</td></tr>
</table>
```

Figure 5.2 shows what Listing 5.17 looks like when you save it as an HTML file and view it in Internet Explorer 5.

FIGURE 5.2

Screenshot of Listing 5.17 viewed in Internet Explorer 5.


E:\XML\05list17.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\XML\05list17.html
Go Links
Done
My Computer

Focus	Ford	2000
Golf	Volkswagen	1999
Camry	Toyota	1999
Civic	Honda	2000
Prizm	Chevrolet	2000

Inserting Attributes with a Generated Name

Earlier you learned that you can insert elements with a name generated with an expression. You can do the same with attributes by using `xsl:attribute`. This element's `name` attribute gives the created attribute its name. Just as with `xsl:element`, this can be literal text, an expression, or a mix of literal text and expressions. Listing 5.18 shows an example with literal text; this example is equivalent to Listing 5.16.

LISTING 5.18 Partial Stylesheet Inserting Literal Attributes

```
<xsl:template match="cars">
  <table>
    <xsl:attribute name="border">1</xsl:attribute>
    <xsl:attribute name="width">500</xsl:attribute>
    <xsl:apply-templates />
  </table>
</xsl:template>

<xsl:template match="car">
  <tr>
    <xsl:attribute name="bgcolor">#dddddd</xsl:attribute>
    <xsl:apply-templates select="@*" />
  </tr>
</xsl:template>
```

ANALYSIS In Listing 5.18, the attributes that were inserted literally in Listing 5.16 have been replaced by `xsl:attribute` elements on lines 3, 4, and 11. The value of the `name` attribute of the `xsl:attribute` element is the name used literally in Listing 5.16. The value of the created attribute is given in between the tags of the `xsl:attribute` element. The result of Listing 5.18 applied to Listing 5.4 would be identical to Listing 5.17 because Listings 5.16 and 5.18 are the same as far as the processor is concerned.

Earlier you looked at some samples that converted Listing 5.4 into an XML document with only elements, as shown in Listing 5.8. With the `xsl:attribute` element, you also can perform this process in reverse, converting Listing 5.8 into Listing 5.4. Listing 5.19 shows a stylesheet that does this.

Note

You should realize by now that there is nearly always more than one way to get the same result. Throughout this book, the samples are used to explain a specific function, but they may not show the most efficient way to perform a task.

LISTING 5.19 Stylesheet Converting Listing 5.8 into Listing 5.4

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

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

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

  <xsl:template match="*">
    <xsl:attribute name="{name()}">
      <xsl:value-of select="." />
    </xsl:attribute>
  </xsl:template>
</xsl:stylesheet>
```

Analysis The template that actually converts attributes to elements starts on line 20. It matches any element, but the `cars` and `car` elements are overridden by the templates on lines 9 and 14, respectively. The match expression for the template on line 20 could also have been `model|manufacturer|year`, but then any other child elements of the `car` element would not be converted, as well as any child elements with different names elsewhere in the document tree. The template creates a new attribute each time it is invoked, with the name and value of the context element.

As you can see from Listing 5.19, the `xsl:attribute` element doesn't necessarily have to be a child element of some element being created. It can exist inside a matched or called template. The attribute is then added to the element that is being created from another template—the `car` element in Listing 5.19. This capability is useful when you need to create multiple related attributes. You can create a named template that inserts the attributes you need, as shown in Listing 5.20.

Listing 5.20 Partial Stylesheet Inserting Attributes from a Named Template

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <table>
11:         <xsl:call-template name="tableprop" />
12:         <xsl:apply-templates />
13:     </table>
14:   </xsl:template>
15:
16:   <xsl:template match="car">
17:     <tr>
18:         <xsl:call-template name="rowprop" />
19:       <xsl:apply-templates select="@*" />
20:     </tr>
21:   </xsl:template>
22:
23:   <xsl:template match="@*">
24:     <td><xsl:value-of select="." /></td>
25:   </xsl:template>
26:
27:   <xsl:template name="tableprop">
28:     <xsl:attribute name="border">1</xsl:attribute>
29:     <xsl:attribute name="width">500</xsl:attribute>
30:   </xsl:template>
31:
```

```
32:   <xsl:template name="rowprop">
33:     <xsl:attribute name="bgcolor">#dddddd</xsl:attribute>
34:   </xsl:template>
35: </xsl:stylesheet>
```

ANALYSIS In Listing 5.20, the two named templates defined on lines 27 and 32 add attributes with properties for HTML `table` elements and `tr` (row) elements, hence their names `tableprop` and `rowprop`. The former is called from line 11 in the `cars` template, which inserts a `table` element and then adds its attributes using the named template. The `rowprop` template is called from line 18 in the `car` element, which adds a table row for each car. If you apply Listing 5.20 to Listing 5.4, the result is similar to Listing 5.17.

In Listing 5.20, it doesn't make much difference if the attributes are inserted using named templates because no other templates call the named templates. So, creating those templates is actually overkill. If multiple templates create elements with identical attributes, using this method would be effective for keeping the attributes in a central place so that they can be edited there if they need to change for all the elements.

Using an Attribute-Set

NEW TERM If you want to insert multiple literal attributes along with an element, using a named (or matched) template is not the only solution. Within a stylesheet, you can define a set of attributes that can be inserted with an element. This set of attributes is appropriately named an *attribute-set* and can be defined using the `xsl:attribute-set` element. You can then insert it by using the `use-attribute-set` attribute with `xsl:element`, as shown in Listing 5.21.

LISTING 5.21 Partial Stylesheet Using an Attribute-Set

```
<xsl:template match="cars">
  <xsl:element name="table" use-attribute-sets="tableprop">
     <xsl:apply-templates />
  </xsl:element>
</xsl:template>

<xsl:template match="car">
  <xsl:element name="tr" use-attribute-sets="rowprop">
    <xsl:apply-templates select="@*" />
  </xsl:element>
</xsl:template>

<xsl:attribute-set name="tableprop">
  <xsl:attribute name="border">1</xsl:attribute>
  <xsl:attribute name="width">500</xsl:attribute>
```

Listing 5.21 Continued

```
16:   </xsl:attribute-set>
17:
18:   <xsl:attribute-set name="rowprop">
19:     <xsl:attribute name="bgcolor">#dddddd</xsl:attribute>
20:   </xsl:attribute-set>
```

Analysis When you apply Listing 5.21 to Listing 5.4, the result is like Listing 5.17. As you can see on lines 13–20, two attribute-sets are defined, similar to the named templates in Listing 5.20. Apart from the fact that they define an attribute-set, not a template, the code is similar. How the attributes are inserted differs more, however. On line 2, the `tableprop` attribute-set is added using the `use-attribute-sets` attribute. The same goes for the `rowprop` attribute-set, which is used on line 8 as part of the `tr` element's definition.

Using Multiple Attribute-Sets

Attribute-sets are flexible. You can employ several methods to use multiple attribute-sets. The easiest way to use them is to specify multiple attribute-sets as a value of the `use-attribute-sets` attribute, separated by whitespace. Listing 5.22 shows a sample.

Listing 5.22 Partial Stylesheet Using Multiple Attribute-Sets

```
1:    <xsl:template match="cars">
2:      <xsl:element name="table" use-attribute-sets="border width">
3:         <xsl:apply-templates />
4:      </xsl:element>
5:
6:    <xsl:attribute-set name="border">
7:      <xsl:attribute name="border">1</xsl:attribute>
8:    </xsl:attribute-set>
9:
10:   <xsl:attribute-set name="width">
11:     <xsl:attribute name="width">500</xsl:attribute>
12:   </xsl:attribute-set>
```

Analysis In Listing 5.22, the `tableprop` attribute-set from Listing 5.21 is replaced by two attribute-sets, `border` and `width`, defined on lines 6 and 10. On line 2, the value of `use-attribute-sets` now contains both these attribute-sets separated by a space. If you apply this stylesheet to Listing 5.4, you again end up with the results shown in Listing 5.17.

The `use-attribute-sets` attribute is not limited to use with `xsl:element`. You can, in fact, create an attribute-set with this attribute, in the process creating an attribute-set that consists of all the attribute-sets listed. This is yet another way to replace the `tableprop` attribute-set in Listing 5.21, as shown in Listing 5.23.

LISTING 5.23 Attribute-Set in an Attribute-Set

```
<xsl:attribute-set name="tableprop" use-attribute-sets="border">
  <xsl:attribute name="width">500</xsl:attribute>
</xsl:attribute-set>

<xsl:attribute-set name="border">
  <xsl:attribute name="border">1</xsl:attribute>
</xsl:attribute-set>
```

ANALYSIS In Listing 5.23, a `tableprop` attribute-set is defined; it uses `use-attribute-sets` to “import” the `border` attribute-set. An attribute also is added, so the result is basically the same as the `tableprop` attribute-set in Listing 5.21. It is not surprising that the result of Listing 5.23 is the same as the result from Listing 5.21; this result is shown in Listing 5.17.

NEW TERM Because you can nest attribute-sets as shown, you can create an elaborate structure of attribute-sets, referring to one another. This capability is useful in situations in which you are creating HTML or XSLFO output with elaborate formatting. You need to be careful, however, that you don’t have a *circular reference,* which occurs when element A refers to element B, which refers back to element A, thus creating a never-ending loop. Circular references can also occur with more elements.

Copying Elements from the Source Document

In some cases, copying elements from the source document into the result without altering them might be useful. In some of the sample listings earlier in this lesson, I did that by re-creating the element or elements in question. A better way to do that is to use other elements, as described in the following sections.

Copying Only the Context Node

NEW TERM The `xsl:copy` element copies the context node from the source to the result. This is called *shallow copy*, so if you want to copy attributes or child elements, you need to explicitly code for this type of copying. Shallow copy means that only the context node is copied. Any attributes or child nodes the node may have are *not* copied along with the node.

Listing 5.24 shows clearly what the `xsl:copy` element does and does not copy.

Listing 5.24 Stylesheet Employing the `xsl:copy` Element

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <xsl:copy>
11:       <xsl:apply-templates />
12:     </xsl:copy>
13:   </xsl:template>
14:
15:   <xsl:template match="car">
16:     <xsl:copy />
17:   </xsl:template>
18: </xsl:stylesheet>
```

Analysis In Listing 5.24, the `xsl:copy` element is used in two different ways. The first one has an opening tag on line 10 and a closing tag on line 12. This method effectively creates opening and closing tags for the `cars` element because it is the context node. The element's value is determined by the result from the `xsl:apply-templates` element on line 11. The `xsl:copy` element on line 16 makes a copy of each `car` element for which the template is invoked. Be aware that this method creates only an empty `car` element, without any of the attributes. The big difference between the two different `xsl:copy` elements in Listing 5.24 is that the value of the element created on line 10 is added specifically. If that isn't done, which is the case on line 16, then the element is empty. You can clearly see this result in Listing 5.25.

Output **Listing 5.25** Result from Applying Listing 5.24 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><cars>
  <car/>
  <car/>
  <car/>
  <car/>
  <car/>
</cars>
```

The `xsl:copy` element supports more or less the same functions that `xsl:element` and `xsl:attribute` elements offer. You can add child elements and attributes just as you would with the latter two, and you also can use the `use-attribute-sets` attribute to add attributes. Because I thoroughly discussed these features with `xsl:element`, I will not discuss them any further. Listing 5.26 shows a sample combining `xsl:copy` with `uses-attribute-sets`. Listing 5.27 shows the result from applying Listing 5.26 to Listing 5.4.

LISTING 5.26 Partial Stylesheet Adding Attributes to Copied Elements

```
1:    <xsl:template match="car">
2:      <xsl:copy use-attribute-sets="year">
3:        <xsl:attribute name="model">Focus</xsl:attribute>
4:      </xsl:copy>
5:    </xsl:template>
6:
7:  <xsl:attribute-set name="year">
8:      <xsl:attribute name="year">1999</xsl:attribute>
9:    </xsl:attribute-set>
```

ANALYSIS In Listing 5.26, an attribute-set is defined on line 7. The `xsl:copy` element on line 2 uses this attribute-set. In addition, line 3 adds another attribute named `model`.

OUTPUT **LISTING 5.27** Result from Applying Listing 5.26 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><cars>
  <car year="1999" model="Focus"/>
  <car year="1999" model="Focus"/>
  <car year="1999" model="Focus"/>
  <car year="1999" model="Focus"/>
  <car year="1999" model="Focus"/>
</cars>
```

ANALYSIS The order in which the attributes appear in Listing 5.27 is deliberate. Any attribute-sets are processed first, in the order in which they are added, and then other attribute definitions are inserted. If an attribute is defined more than once, the last one added is the one sent to the output.

Copying Node-Sets and Tree Fragments

NEW TERM The `xsl:copy` element is useful, but it operates only on the context node and copies only the context node, without any of the attributes or child elements. Although you can get around both deficiencies by creating several templates, another element can copy based on an XPath expression and performs *deep copy,* which is the opposite of shallow copy. In this type of copy, the selected node or node-set and all attributes and descendant elements are copied. In effect, the entire tree fragment under the node or under each node in the selected node-set is copied to the output.

You use the `xsl:copy-of` element to perform a deep copy. Unlike the `xsl:copy` element, this element is always empty and has a mandatory `select` attribute. The `select` attribute's value must be a valid XPath expression selecting a node or node-set. Listing 5.28 shows a simple usage of `xsl:copy-of`.

LISTING 5.28 Partial Stylesheet Copying the Entire Document

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:apply-templates />
7:    </xsl:template>
8:
9:    <xsl:template match="cars">
10:     <xsl:copy-of select="." />
11:   </xsl:template>
12: </xsl:stylesheet>
```

ANALYSIS Line 10 in Listing 5.28 selects the context node and copies it to the output, including all its attributes and descendant elements. Because the context node is the `cars` element, which is effectively the root element, the entire document is copied to the output. So, if you apply Listing 5.28 to Listing 5.4, the result is the same as Listing 5.4. Because `xsl:copy-of` is used in the template matching the `cars` element, only a fragment of the document is copied to the output if the `cars` element is not the root element of the source document.

If the select expression yields a node-set, for each node in the node-set, the entire tree fragment is copied. This goes for any node-set, even node-sets composed of nodes from different sections in the tree. As such, the node-set should be used with care because you may end up copying many elements you aren't supposed to. Listing 5.29 shows a replacement template for the template on line 9 of Listing 5.28. The template in Listing 5.29 copies a node-set of two cars.

LISTING 5.29 Partial Stylesheet Copying a Node-Set

```
<xsl:template match="cars">
  <xsl:copy>
```

```
    <xsl:copy-of select="car[1]|car[3]" />
  </xsl:copy>
</xsl:template>
```

The result from applying Listing 5.28 with the changes from Listing 5.29 to Listing 5.4 is shown in Listing 5.30.

OUTPUT **LISTING 5.30** Result from Applying Listing 5.29 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><cars>
➥<car model="Focus" manufacturer="Ford" year="2000"/>
➥<car model="Camry" manufacturer="Toyota" year="1999"/></cars>
```

ANALYSIS As you can see in Listing 5.30, Listing 5.29 copies only the first and third `car` elements from Listing 5.4. It does, however, copy them with all their attributes.

Using both the `xsl:copy` and `xsl:copy-of` elements, you can copy large parts of documents very selectively. The `xsl:copy` element requires more work than the `xsl:copy-of` element, but it gives you more control over the output.

Inserting Comments and Processing Instructions

Comments and processing instructions perform a special function in an XML document. Comments are mostly meant for a human reader, telling him or her what is going on in a document. In HTML, comments may also serve to let older browsers ignore a piece of code that they will not understand, such as JavaScript code. Processing instructions are the opposite of comments. They are not meant for the reader; instead, they are instructions for the program reading the XML document. For instance, in XML, a processing instruction is used to attach a stylesheet to an XML document, so if it is viewed by an XML/XSLT-enabled viewer, the viewer will apply the stylesheet automatically. CSS documents are attached to HTML documents the same way.

Inserting Comments

Because comments are not actually part of the XML structure, you can insert them almost anywhere in a document. They can appear at the start or end of a document, beyond the root element of the document, and anywhere a text value is also possible. This does mean that you can't insert a comment within a start or end tag.

You can insert a comment by using the `xsl:comment` element. This element has no attributes, and the element value is output as the comment. Now consider this example:

```
<xsl:comment>This is a comment</xsl:comment>
```

In XML or HTML output, this code results in the following being inserted into the output at the location the comment was inserted:

```
<!--This is a comment-->
```

The `<!--` character sequence denotes the start of a comment; the `-->` sequence, the end, just like start and end tags of an XML element. Because of this, it is not legal to have `--` in the comment itself; using these characters would result in an error.

Comments are very useful when you're debugging a stylesheet. For instance, inserting a comment at the start of each template shows you exactly which templates are matched for a source document. Because the comments don't interfere with the XML, you can use the reulting XML in other applications just as before. Listing 5.31 shows Listing 5.24 with debugging comments.

LISTING 5.31 Listing 5.24 with Comments Added to Templates

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:comment>template match="/"</xsl:comment>
7:      <xsl:apply-templates />
8:    </xsl:template>
9:
10:   <xsl:template match="cars">
11:     <xsl:comment>template match="cars"</xsl:comment>
12:     <xsl:copy>
13:       <xsl:apply-templates />
14:     </xsl:copy>
15:   </xsl:template>
16:
17:   <xsl:template match="car">
18:     <xsl:comment>template match="car"</xsl:comment>
19:       <xsl:copy />
20:   </xsl:template>
21: </xsl:stylesheet>
22: </xsl:stylesheet>
```

ANALYSIS Lines 6, 11, and 18 of Listing 5.31 insert a comment when the templates on lines 5, 10, and 17 are invoked. This way, you get a complete record of which templates are matched and in what sequence. The result is shown in Listing 5.32.

OUTPUT **LISTING 5.32** Result from Applying Listing 5.32 to Listing 5.4

```
1: <?xml version="1.0" encoding="utf-8"?><!--template match="/"-->
➥<!--template match="cars"--><cars>
2:   <!--template match="car"--><car/>
3:   <!--template match="car"--><car/>
4:   <!--template match="car"--><car/>
5:   <!--template match="car"--><car/>
6:   <!--template match="car"--><car/>
7: </cars>
```

ANALYSIS On line 1 of Listing 5.32, you can see that the root template is matched first and only once. Then the `cars` template is matched, followed by a match of the `car` template for each car element in Listing 5.4. The comments show all this clearly, but you can still use this document as if it is the result in Listing 5.25.

A comment doesn't have to contain literal text. You can use XSLT elements such as `xsl:value-of` to create the value. This allows you to see which element is being operated on or to provide other useful information.

Inserting Processing Instructions

A processing instruction is used to give a program reading a document additional information on how to process the document. These instructions can be very specific and program/vendor-related instructions or common instructions, such as attaching a stylesheet to an XML or HTML file. Because processing instructions often have bearing on the entire document, they should be inserted at the start of a document. Processing instructions are not widely used in XML, but in HTML, they are more common. It is therefore good to know how to insert them.

Inserting a processing instruction is just as easy as inserting an element, except that you insert a processing instruction with the `xsl:processing-instruction` element. Like `xsl:element`, this element has a mandatory `name` attribute used to identify the processing instruction. The element value is used as the value of the processing instruction. Listing 5.33 shows a sample using a processing instruction.

LISTING 5.33 Stylesheet Inserting a Processing Instruction

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:processing-instruction name="xml-stylesheet">href="05list05.xsl"
➥type="text/xml"</xsl:processing-instruction>
```

LISTING 5.33 Continued

```
7:    <xsl:copy-of select="." />
8:   </xsl:template>
9: </xsl:stylesheet>
```

ANALYSIS On line 6 of Listing 5.33, a processing instruction is inserted to attach Listing 5.5 to it as a stylesheet. The name given to the processing instruction is therefore `xml-stylesheet`. The value links to the file containing Listing 5.5 and specifies that that file is XML. This is just like the processing instruction used on Day 2, "Transforming Your First XML," to attach a stylesheet to an XML source. Line 7 copies the entire document in Listing 5.4 to the output, so the result is as shown in Listing 5.34.

OUTPUT **LISTING 5.34** Result from Applying Listing 5.33 to Listing 5.4

```
<?xml version="1.0" encoding="utf-8"?><?xml-stylesheet href="05list05.xsl"
➥ type="text/xml"?><cars>
  <car model="Focus" manufacturer="Ford" year="2000"/>
  <car model="Golf" manufacturer="Volkswagen" year="1999"/>
  <car model="Camry" manufacturer="Toyota" year="1999"/>
  <car model="Civic" manufacturer="Honda" year="2000"/>
  <car model="Prizm" manufacturer="Chevrolet" year="2000"/>
</cars>
```

ANALYSIS Listing 5.34 displays the same document as Listing 5.4. The difference is in the first line, which contains a processing instruction attaching Listing 5.5 to it. If you viewed Listing 5.34 in an XML/XSLT enabled browser, it would appear as Listing 5.6 displayed in a browser.

Summary

Today you learned all about inserting text, elements, and attributes. Although most methods are fairly easy, you can choose from many methods. If you need to insert literal elements and attributes, the easiest way is to write them in the stylesheet as they should appear. When they need to be generated from data in the source document, you use `xsl:element` and `xsl:attribute`.

You can quickly add sets of elements and attributes using named (or matched) templates. If you need to insert sets of attributes, you have the option of using the `use-attribute-sets` attribute to specify sets of predefined attributes that need to be inserted. Attribute-sets can be created from other attribute-sets, so you can create quick and easy inserts that actually insert a complex set of attributes. If two attributes inserted for an element are the same, the one defined last prevails.

Tomorrow you will learn more details about taking action based on elements or attributes in the source document. This way, you can insert elements and text even more selectively.

Q&A

Q I have seen several methods to insert elements and attributes. Which is best?

A There is not really a best way to insert elements and attributes. The XSLT specification does not include any implementation details for the processors, so different processors use different implementations with different performance characteristics. Go with what works for you.

Q Why doesn't `xsl:copy-of` support attribute-sets?

A Suppose you copy a large tree fragment. In this case, attributes are added to all elements in the tree fragment. The likelihood that you would want this result is very small. When you need this result, you can create a named template to add the attributes.

Q Does `xsl:copy-of` have to operate on the context node?

A Certainly not. The `select` attribute can contain an expression that selects any node or node-set. It does not have to be the context node or relative to the context node.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is very helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: For an XSLT processor, there is a difference between inserting characters as is and using output escaping.
2. True or False: You can use `xsl:attribute` only to create attributes for an element created with `xsl:element`.
3. What is the purpose of the XPath function `name()`?
4. What is the difference between shallow copy and deep copy?
5. Is there a difference between `<xsl:element name="{name()}" />` and `<xsl:copy />`?

Exercise

1. Create a stylesheet for Listing 5.4 to create the output from Listing 5.8, but use `xsl:copy` and `xsl:element` where possible.

This page intentionally left blank

WEEK 1

DAY 6

Conditional and Iterative Processing

Yesterday you learned how to insert text, elements, and attributes into the resulting output. You also learned how to copy elements from the source document into the resulting output. These skills are important when you're transforming XML because you now can create nearly all sorts of output from an element or attribute.

Also key to the transformation process is the flow of control. Currently, you can influence it—and thus what the output looks like—through the use of templates. Although templates are powerful, they are not always the best tool when you need to differentiate between nodes to be inserted.

The focus of today's lesson is XSLT functionality that enables you to selectively act on elements within an invoked template.

Today you will learn the following:

- How to iterate through a node-set
- How to act on nodes based on the data value of those nodes

- How to create conditional expressions
- More details about filtering nodes based on predicate expressions

Iterating Through a Node-Set

New Term So far, the only way to control the flow of control has been to use templates. Templates are powerful and flexible but also relatively hard to work with because processing is driven by the source document's data and the flow of control is therefore hard to predict. This is the nature of *push processing,* in which something outside the program determines the next action to be taken. In effect, the program reacts to some event.

New Term Push processing is mainly responsible for the power of XSLT, but in some cases, it is hard to use and read. Much easier to use and much easier to follow when it comes to flow of control is *pull processing,* in which a program retrieves data and acts on it. This means that the program determines the next action to be taken. This process is called pull processing because the program "pulls in the data."

A perfect example of push processing occurs when templates are matched. When you output a value using the `xsl:value-of` element, you use a process that is more or less pull processing, especially if you select a value from an element or attribute using absolute addressing. Although this form of pull processing is quite handy, it isn't useful when you want to select a node-set and do something with it. The `xsl:for-each` element comes into play here. The `xsl:for-each` element enables you to iterate through a node-set and act on the nodes in the node-set separately. This element is much like a template, but it uses pull processing instead of push processing.

Processing Each Node in a Node-Set

Most programming languages, such as C, Java, and Visual Basic, contain a statement that can be used to iterate or loop a predetermined number of times. This statement is called a `for` loop; in C and Java, it looks like this:

```
for(var i = 1; i <= 10;i++) {
    //execute some code
}
```

The code is based on a counter denoted by the variable `i`. It starts off with the value 1, and each time the code in between the curly braces is executed, its value is increased by 1. This process goes on as long as the value is 10 or lower. In Visual Basic (or VBScript), the same code would look like this:

```
For i = 1 To 10
    'execute some code
Next
```

The Visual Basic code is somewhat easier to read if you're not familiar with programming, yet it does exactly the same thing. The `Next` statement serves the same purpose as the ending curly brace in C or Java: It denotes the end of the code block to be executed with each iteration.

The problem with these constructs is that they iterate a predetermined number of times, and such code is not very handy when you're using collections or sets of data. In that case, you just want to iterate through each item in the collection, no questions asked, without having to determine the number of items in the collection. In Visual Basic, this problem has been solved by adding another type of `for` loop:

```
For Each x In MyCollection
    'execute some code with x
Next
```

The `For Each ... In ...` statement is almost declarative programming because it tells the program to do something with each item in the collection; how each item is selected is up to the program. You can use this approach because you are working with a collection, which by its very nature doesn't have any particular order to the items it contains. This also goes for node-sets because they are processed as the processor encounters them.

Note

Manipulating the order in which nodes are processed will be discussed on Day 12, "Sorting and Numbering."

XSLT has a construct that is similar (if not the same) as Visual Basic's `For Each ... In ...` statement: the `xsl:for-each` element. Just like a match template, this element has a `select` attribute with which you select the elements that need to be processed using an XPath expression. The best way to show you how this element works is to use an example. The sample XML source in Listing 6.1 is based on an earlier sample.

LISTING 6.1 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <employee name="Joe Dishwasher">
    <phonenumber type="home">555-1234</phonenumber>
    <phonenumber type="fax">555-2345</phonenumber>
    <phonenumber type="mobile">555-3456</phonenumber>
  </employee>
  <employee name="Carol Waitress">
    <phonenumber type="home">555-5432</phonenumber>
```

Listing 6.1 Continued

```
      <phonenumber type="mobile">555-9876</phonenumber>
    </employee>
  </employees>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

So far, all transformations have been performed based on templates. Listing 6.2 also creates output from Listing 6.1, but does so by using the `xsl:for-each` element instead of a template-based approach with matching.

Listing 6.2 Stylesheet for Listing 6.1 Using `xsl:for-each`

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:for-each select="employees/employee">
7:        *<xsl:value-of select="@name" />*
8:        <xsl:for-each select="phonenumber">
9:          -<xsl:value-of select="@type" />: <xsl:value-of select="." />
10:       </xsl:for-each>
11:     </xsl:for-each>
12:   </xsl:template>
13: </xsl:stylesheet>
```

Analysis Listing 6.2 has only one template, which matches the document root. Instead of using `xsl:apply-templates`, the template contains `xsl:for-each` elements on lines 6 and 8 to dive deeper into the source's XML document tree. The `xsl:for-each` element's select expression on line 6 selects all the `employee` elements that are children of the `employees` element. This is a major difference with templates. A template matching the `employee` element will match any `employee` element, no matter where in the document it is. The `xsl:for-each` element selects only elements relative to the context node, unless instructed otherwise—for instance, using `//employee`, which selects all the `employee` elements in the entire source document.

With each `employee` node in the processed node-set, line 7 inserts the name of the employee. The `xsl:for-each` element on line 8 iterates through all the `phonenumber` elements that are children of the current employee and outputs the `type` attribute and value. Listing 6.3 shows the result from applying Listing 6.2 to Listing 6.1.

OUTPUT **LISTING 6.3** Result from Applying Listing 6.2 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*

        -home: 555-1234
        -fax: 555-2345
        -mobile: 555-3456
      *Carol Waitress*

        -home: 555-5432
        -mobile: 555-9876
```

Note

The * and - inserted by the stylesheet in Listing 6.3 make the output easier to read. These characters serve no other function.

In Listing 6.2, you can easily determine what the context node is before you even run the code. With each `xsl:for-each` element, you dig deeper into the document tree, iterating through child elements of the element that is currently iterated by a higher level `xsl:for-each` element. Because you specifically select the nodes to be processed, there are no side effects, something that is common with templates.

Caution

The `xsl:for-each` element cannot exist outside a template and therefore cannot serve as a replacement for a template. Of the elements discussed so far, only the `xsl:template` element can be used as a child element of `xsl:stylesheet`. You need to place the other elements inside a template. Also, remember that you cannot nest templates.

Listing 6.2 might make you think that you can use only one template if you're using `xsl:for-each`. That is not the case. You can mix the push processing of templates with the pull processing of `xsl:for-each`. The stylesheet in Listing 6.4 employs two templates and an `xsl:for-each` element.

LISTING 6.4 Stylesheet with Multiple Templates and Iteration

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:for-each select="employees/employee">
```

LISTING 6.4 Continued

```
7:        *<xsl:value-of select="@name" />*
8:        <xsl:apply-templates />
9:      </xsl:for-each>
10:   </xsl:template>
11:
12:   <xsl:template match="phonenumber">
13:     -<xsl:value-of select="@type" />: <xsl:value-of select="." />
14:   </xsl:template>
15: </xsl:stylesheet>
```

ANALYSIS Applying Listing 6.4 to Listing 6.1 produces nearly the same result as Listing 6.2. The only difference is the whitespace that is inserted because of the indentation of the stylesheet elements. Yet, the two stylesheets are rather different. Instead of using `xsl:for-each` to iterate through the `phonenumber` elements as on line 8 of Listing 6.2, Listing 6.4 uses a match template on line 12, in the process transferring from pull processing to push processing.

Filtering Node-Sets

In most stylesheets used in this and previous lessons, either all nodes in a node-set were processed, or none of them were. You have learned only how to manipulate them using position predicates that tell the processor which node it should get from the source XML, based on its position within the document. More often, you will want to operate on certain nodes, based on their name or value, and leave the other nodes. You can do so by using predicates that go beyond position filtering. These predicates can be used to compare the value of some node to the value you need it to be. Also, you can compare node names, check whether certain nodes exist, and so on.

NEW TERM When you use predicates to filter nodes in a node-set, the expression used in the predicates has to be evaluated and result in the values `true` or `false`. If the value is `true`, the node being evaluated will be processed; if the value is `false`, the node will be ignored. The values `true` and `false` are said to be *Boolean* values. Values of the Boolean data type can be only `true` or `false`, 1 or 0, on or off. In the case of XSLT, the values are `true` or `false`.

An expression's resulting value isn't always a Boolean value. It also can be a number, string, node-set, or tree fragment. In those cases, the expression's resulting value is converted to Boolean. This topic will be covered later; for now, a simple example will help you understand predicate expressions. In Listing 6.5, a predicate is used with a comparison between the value of the `type` attribute and literal value `'home'`.

LISTING 6.5 Iteration Using a Filter Expression

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:template match="/">
 6:     <xsl:for-each select="employees/employee">
 7:       *<xsl:value-of select="@name" />*
 8:       <xsl:for-each select="phonenumber[@type='home']">
 9:         <xsl:value-of select="@type" />: <xsl:value-of select="." />
10:       </xsl:for-each>
11:     </xsl:for-each>
12:   </xsl:template>
13: </xsl:stylesheet>
```

ANALYSIS On line 8 in Listing 6.5, a predicate filters the `phonenumber` elements. If the value of the `type` attribute is `'home'`, the node is processed and thus written into the resulting output. Hence, the listing will output only the employees' home phone numbers and ignore any other type of phone number, as you can see in Listing 6.6.

OUTPUT **LISTING 6.6** Result from Applying Listing 6.5 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*
      home: 555-1234
      *Carol Waitress*
      home: 555-5432
```

ANALYSIS In Listing 6.6, no side effects occur from filtering the node-set. The `phonenumber` elements that don't satisfy the condition of the predicate are ignored completely.

If you were to use a template to match the `phonenumber` elements rather than select them, you would see the difference between matching and selecting. In Listing 6.7, the `xsl:for-each` element is replaced by a template.

LISTING 6.7 Template Using a Filter Expression

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:template match="/">
 6:     <xsl:for-each select="employees/employee">
 7:       *<xsl:value-of select="@name" />*
 8:       <xsl:apply-templates select="*" />
```

LISTING 6.7 Continued

```
9:      </xsl:for-each>
10:   </xsl:template>
11:
12:   <xsl:template match="phonenumber[@type='home']">
13:     <xsl:value-of select="@type" />: <xsl:value-of select="." />
14:   </xsl:template>
15: </xsl:stylesheet>
```

ANALYSIS Listing 6.7 uses a template on line 12 instead if the `xsl:for-each` element on line 8 of Listing 6.5. Both are meant to operate on a specific `phonenumber` element. You might think that Listing 6.7 will yield the same result as Listing 6.5. There is a catch, however, because the elements are matched rather than selected. The actual result is shown in Listing 6.8.

OUTPUT **LISTING 6.8** Result from Applying Listing 6.7 to Listing 6.1

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:        *Joe Dishwasher*
3:        home: 555-1234555-2345555-3456
4:        *Carol Waitress*
5:        home: 555-5432555-9876
```

ANALYSIS If you look closely at lines 3 and 5, you can see that the phone numbers are not correct. They actually consist of the values of all the employees' `phonenumber` elements. The home phone number is matched correctly by the template on line 12 of Listing 6.7 and has the correct output. The other `phonenumber` elements are not matched because the matching rule in the template applies only to home phone numbers. Therefore these elements are matched by the built-in template rule. This is why their value is written to be output. Because the elements are matched instead of selected, a side effect occurs for the nodes that aren't matched. This is exactly what makes writing style sheets on the basis of matching difficult. You have to be very aware of any side effects that may occur and preemptively handle them.

Using Node-Set Functions

On Day 3, "Selecting Data," you learned that you can use a predicate with the number of the node you want to select from a node-set. This number is the position that the node has within that node-set. This position is partially determined by the order of the nodes in the source document. Listing 6.9 should jog your memory if you don't remember.

LISTING 6.9 Stylesheet Using a Numeric Predicate

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:for-each select="employees/employee/phonenumber[2]">
7:        *<xsl:value-of select="../@name" />*
8:        <xsl:value-of select="@type" />: <xsl:value-of select="." />
9:      </xsl:for-each>
10:   </xsl:template>
11: </xsl:stylesheet>
```

When you apply Listing 6.9 to Listing 6.1, you get the result in Listing 6.10.

OUTPUT **LISTING 6.10** Result from Applying Listing 6.9 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*
      fax: 555-2345
      *Carol Waitress*
      mobile: 555-9876
```

ANALYSIS In Listing 6.10, only the `phonenumber` elements that were second in their respective node-sets are shown. Line 6 in Listing 6.9 ensures that this number is shown, using the predicate expression `[2]`. But wait a minute. Shouldn't the value of the predicate expression yield a Boolean value? It should indeed, and it does, although not explicitly. Because only a numeric value is available, XSLT assumes that you actually meant the following:

```
<xsl:for-each select="employees/employee/phonenumber[position()=2]">
```

XSLT assumes that you want to compare the numeric value to the `position()` function. This function returns the numeric position of the context node within the current node-set. For the first node, the resulting expression is `1=2`, which yields `false`. For the second node, the actual expression becomes `2=2`, which is `true`; hence, the node is processed.

XSLT contains another function that has bearing on the current context: `last()`.It returns the numeric position of the last node within the current node-set. If you change line 6 in Listing 6.9 to

```
<xsl:for-each select="employees/employee/phonenumber[last()]">
```

it will process only the last `phonenumber` element of each employee's `phonenumber` node-set. Here again, XSLT assumes that `last()` actually means `position()=last()` because the `last()` function returns a numeric value. Listing 6.11 shows the result if you exchange the preceding line for line 6 in Listing 6.9.

OUTPUT

LISTING 6.11 Result from Stylesheet Using `last()`

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*
      mobile: 555-3456
      *Carol Waitress*
      mobile: 555-9876
```

ANALYSIS Listing 6.11 shows the correct mobile number of both employees. For both employees, this number is the last `phonenumber` element in the node-set. In the case of Joe Dishwasher, it is the third element, and in the case of Carol Waitress, the second. So, the `last()` function returns `3` and `2`, respectively, which are then compared to the position of the current node.

Conditional Processing

NEW TERM You have learned how to filter out nodes that you don't want, either through matching or by selection. You therefore can process those nodes you want and act on them as necessary to create the output you want. What you haven't learned yet is how to insert data based on some condition. Filtering using predicates comes close, but many things you can do with conditional processing are very hard using predicate expressions and templates or iteration. In *conditional processing,* a piece of code is executed if some condition holds true.

Conditional processing comes in two forms. One allows you to execute a piece of code based on a condition. The other allows you to select a piece of code to execute from multiple alternatives.

Simple Conditional Processing

The easiest form of conditional processing uses the `xsl:if` element, which has one attribute named `test`. This attribute's value needs to be an expression that yields a Boolean value. If the resulting value is `true`, the body of the element is executed; otherwise, it is not. Listing 6.12 shows the `xsl:if` element in action.

LISTING 6.12 Stylesheet Using `xsl:if`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

```

```
5:    <xsl:template match="/">
6:      employees:
7:      <xsl:for-each select="employees/employee">
8:        <xsl:value-of select="@name" />
9:        <xsl:if test="position() != last()">, </xsl:if>
10:       <xsl:if test="position() = last()">.</xsl:if>
11:     </xsl:for-each>
12:   </xsl:template>
13: </xsl:stylesheet>
```

ANALYSIS Line 7 of Listing 6.12 selects all `employee` elements in Listing 6.1 and iterates through them. On line 8, the value of the `name` attribute is sent to the output. Lines 9 and 10 both contain an `xsl:if` element to see what needs to be inserted next. Line 9 checks whether the current node is *not* the last node and, if that is true, inserts a comma. Line 10 checks whether the current node is the last node and, if so, inserts a period. The result is shown in Listing 6.13.

OUTPUT **LISTING 6.13** Result from Applying Listing 6.12 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
    employees:
    Joe Dishwasher, Carol Waitress.
```

ANALYSIS Listing 6.13 contains a simple list of the employees in Listing 6.1. The employees' names are separated by commas, and the list ends with a period. No big deal right? Now try to imagine creating this list without using templates or `xsl:for-each`. You cannot insert a comma with each element; if you do, the list would end in a comma instead of a period. Also, you want to insert a period only after the last element. With templates, you would probably end up writing a separate template for the last element. The approach in Listing 6.12 is much easier, requiring two lines of code.

Now look closer at line 10. As I said, it checks whether the context node is the last node in the node-set. If it is, a period is inserted. It does so by comparing the value returned by the `position()` function with that returned by the `last()` function.

Caution

In match and select expressions with predicates, `last()` implies a `position()=last()` comparison. With `test` expressions, this is not the case. The `last()` function returns a number that is converted to a Boolean value, with `0` being `false` and any other value being `true`.

Now look at line 9. Instead of comparing the position and the last position using `=`, this line uses `!=`. The former, of course, checks whether the values on the left and right sides are equal. The latter checks the opposite and returns `true` if the values left and right of it are not equal. Hence, the comma is inserted with each element that is *not* the last element.

You can use several operators when you need to compare two values. They are shown in Table 6.1.

TABLE 6.1 Comparison Operators

Operators	*Description*
`=`	Equal
`!=`	Not equal
`&lt;`	Less than
`&lt;=`	Less than or equal
`&gt;`	Greater than
`&gt;=`	Greater than or equal

The operators in Table 6.1 always need to appear in between the values you want to compare. For the last four operators, you need to read from left to right. An expression such as `A &gt; B` returns `true` if the value of `A` is greater than the value of `B`. These operators may look a little funny, but remember that XML uses certain special characters. Therefore, writing `A > B` would result in an error. To get around this problem, you can output escape the operators where necessary.

Line 9 in Listing 6.12 could have been written differently using another operator. For instance,

```
<xsl:if test="position() &lt; last()">, </xsl:if>
```

yields the same result. The `position()` function's result can, of course, never be greater than the `last()` function's result, but if it could, the test expression given here would actually be more specific. The `!=` operator just forces inequality, but it may be less than or greater than.

Listing 6.9 used a filter so that only the second phone number of each employee was shown. Now look at Listing 6.14, which uses `xsl:if`.

LISTING 6.14 Filtering Nodes with `xsl:if`

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="/">
6:      <xsl:for-each select="employees/employee/phonenumber">
7:        <xsl:if test="position()=2">
8:          *<xsl:value-of select="../@name" />*
9:          <xsl:value-of select="@type" />: <xsl:value-of select="." />
10:       </xsl:if>
11:     </xsl:for-each>
12:   </xsl:template>
13: </xsl:stylesheet>
```

ANALYSIS When you apply Listing 6.14 to Listing 6.1, it does not yield the same result as Listing 6.9. In Listing 6.9, the filter expression that filters the node-sets sees the phone numbers of each employee as separate node-sets. Line 6 in Listing 6.14, however, makes them into one node-set. So, the current context of line 7 is a node-set of all employees' `phonenumber` elements. It therefore tests positive on only one node. Its output is shown in Listing 6.15.

OUTPUT **LISTING 6.15** Result from Applying Listing 6.14 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
        *Joe Dishwasher*
        fax: 555-2345
```

ANALYSIS You can achieve the result in Listing 6.15 if you modify the filter expression on line 6 in Listing 6.9. Changing it to read `employees/employee[1]/phonenumber[2]` would work, but there is a big difference in the way everything is processed. In Listing 6.14, each `phonenumber` element is processed. Line 7 sees to it that no output is generated for it unless it is the second node. When you use the filter expression, only one node is processed.

In most processors, filtering performs better than using conditions because fewer nodes are actually processed. This does not mean, however, that conditional processing is not the preferred method of attack in some situations. Listing 6.12 is a perfect example in which conditional processing is preferred over filtering with templates or selection.

Conditional Processing with Multiple Options

The `xsl:if` element enables you to test whether a certain condition is satisfied, and if it is, to execute code. What happens if you have several blocks of code and only one needs to be executed, based on a given condition? In that case, you need to create a chain of `xsl:if` elements, each testing a condition. The catch is that if more than one of those elements has an expression that tests positive, the code within each element is executed. If you want all the code executed, this outcome is fine, but if only one of the code blocks needs to be executed, you have a problem.

In XSLT, you can use a structure of elements to resolve this problem. The main element is `xsl:choose`. This element has two possible child elements: `xsl:when` and `xsl:otherwise`. The former is similar to the `xsl:if` element in that it contains code to be executed if the expression in the `test` attribute returns `true`. The `xsl:when` element can occur more than once as a child of the `xsl:choose` element, but only one of them is executed. The first element for which the expression returns `true` is executed. Any others that also satisfy their condition are ignored. If no `xsl:when` elements are executed, the `xsl:otherwise` element comes into play. Any code that it contains is executed if none of the `xsl:when` elements are executed. Listing 6.16 shows this structure of elements in action.

LISTING 6.16 Stylesheet Using `choose-when-otherwise`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:for-each select="employees/employee">
      *<xsl:value-of select="@name" />*
        <xsl:for-each select="phonenumber">
          <xsl:choose>
            <xsl:when test="@type='fax'">
              Fax:    <xsl:value-of select="." />
            </xsl:when>
            <xsl:when test="@type='mobile'">
              Mobile: <xsl:value-of select="." />
            </xsl:when>
            <xsl:otherwise>
              Phone:  <xsl:value-of select="." />
            </xsl:otherwise>
          </xsl:choose>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS In Listing 6.16, the `xsl:choose` element starting on line 9 has several `xsl:when` child elements and an `xsl:otherwise` child element. The `xsl:when` elements check to see whether the `type` attribute of the element being processed has a certain value. If it does, the code within the `xsl:when` element is executed. If both `xsl:when` elements do not satisfy the condition of their `test` attribute, the `xsl:otherwise` element on line 16 kicks in.

Note In Listing 6.16, the `xsl:choose` element is a child element of an `xsl:for-each` element. Using the element this way is not mandatory; instead, it can be a child element of a template. In fact, of the elements covered so far, only the `xsl:stylesheet` and `xsl:template` elements have very strict rules. The other elements can be contained in another element. You therefore can have an `xsl:choose` element inside an `xsl:for-each` element, which in turn is a child of an `xsl:if` element.

Listing 6.17 shows the result from applying Listing 6.16 to Listing 6.1.

OUTPUT **LISTING 6.17** Result from Applying Listing 6.16 to Listing 6.1

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:        *Joe Dishwasher*
3:
4:                Phone:  555-1234
5:                Fax:    555-2345
6:                Mobile: 555-3456
7:        *Carol Waitress*
8:
9:                Phone:  555-5432
10:               Mobile: 555-9876
```

ANALYSIS In earlier listings, the phone numbers were preceded by the value of each `phonenumber` element's `type` attribute. In Listing 6.17, you can see that this has been changed to text added in the stylesheet. Added effort has been put into making the numbers line up neatly. This was all possible through the use of `xsl:choose`. Line 5 in Listing 6.17 is inserted by line 11 in Listing 6.16. Notice the whitespace between `Fax:` and the `xsl:value-of` element, which is copied as is to the output. When the `type` attribute's value is neither `fax` nor `mobile`, the `xsl:otherwise` element on line 16 of Listing 6.16 is invoked. This element outputs the text `Phone:` and the phone number. If a `phonenumber` element with a type not currently used in Listing 6.1 were included in the input, that element would also be written as if it were a regular phone number. In Listing 6.18, that is not the case.

LISTING 6.18 `xsl:choose` Element Without `xsl:otherwise`

```
1:  <xsl:choose>
2:    <xsl:when test="@type='fax'">
3:      Fax:    <xsl:value-of select="." />
4:    </xsl:when>
5:    <xsl:when test="@type='mobile'">
6:      Mobile: <xsl:value-of select="." />
7:    </xsl:when>
8:    <xsl:when test="@type='home'">
9:      Phone:  <xsl:value-of select="." />
10:   </xsl:when>
11: </xsl:choose>
```

ANALYSIS Listings 6.18 and 6.16 produce the same output for Listing 6.1; this output is shown in Listing 6.17. However, because the `xsl:otherwise` element of Listing 6.16 is removed and replaced by the `xsl:when` element on line 8, any `phonenumber` element that is not of the `home`, `fax` or `mobile` type will not be displayed at all. As I just explained, the `xsl:otherwise` element would have displayed those elements. So, as you can see, having an `xsl:otherwise` element is by no means necessary, but it is useful for dealing with cases that aren't handled by any of the `xsl:when` elements. If you don't need to do anything in such situations, just leave out the `xsl:otherwise` attribute.

More About Expressions

On Day 3, you learned how to use XPath to match or select a node or node-set. Without going into the details further, you learned about predicates that make a comparison to filter out nodes. At this point, it is useful to recap, formalize, and expand on your knowledge of expressions.

Formalizing Expressions

Expressions are somewhat circular in nature. They contain elements that themselves can contain expressions. Dissecting a sample expression therefore gives you the best view of what an expression is and how it is built up. Look at the following example:

```
/employees/employee[position()=1]/child::phonenumber[attribute::type='home']
```

NEW TERM This match or select expression can be used in a template or with an `xsl:for-each` element. It is called a *location path* because it points to specific nodes in the node tree. These nodes are in the specified location and also satisfy the conditions given by the location path. You can see that the location path uses absolute addressing, because it starts with `/`. The location path consists of several *location steps*, separated by `/`. Each location step consists of the following:

- An optional axis
- A node test
- Zero or more predicates

In the preceding example, `child` specifies the axis of the third location step. It is separate from the node test with the double colon. In the other two location steps, the axes are implied (also `child`). The axis is followed by the node test, which specifies the names of the nodes that actually match on the specified axis. For the separate location steps, they are, in turn, `employees`, `employee`, and `phonenumber`.

So far, this description is very straightforward. The real complexity comes from predicates, which you use to make a more precise match or selection. The expression defined within a predicate always has to return a Boolean value. If that value is `true`, the node is processed further; if `false`, it is ignored. In most cases, a predicate expression is a comparison of two values. The values are compared using the operators in Table 6.1. Comparisons can be made

1. Between the value of a node and a literal value
2. Between the value of a node and the value returned by a function
3. Between the value of a node and the value of another node
4. Between the value returned by a function and a literal value
5. Between the value returned by a function and the value returned by another function

`position()=1` is an example of the fourth option; `attribute::type='home'` is an example of the first option. Predicates can themselves contain entire location paths, including predicates, so the following location path is perfectly legal:

```
employee/phonenumber[@type=/employees/employee[1]/phonenumber[3]/@type]
```

This location path compares the value of the context element's `type` attribute to that of the same attribute of the third `phonenumber` element that is a child element of the first `employee` element. Although this sample is rather strange, you will find plenty of situations in which this sort of comparison makes sense.

Note

You can download a tool called XPath Visualizer from `http://www.topxml.com/xpathvisualizer/default.asp`. This tool can help you build expressions.

Using Multiple Predicates

You are not limited to using just one expression. You can actually test more than one thing just by adding extra predicates. In that case, the node is processed only if all predicates return `true`. Such an expression would look like this:

```
phonenumber[position()=1][@type='home']
```

The preceding expression matches only `phonenumber` elements that are in the first position of the node-set currently being evaluated and that have a `type` attribute with the value `home`. You can add as many attributes as you like.

Combining Expressions

Instead of adding multiple predicates, you also can combine expressions. The expression from the preceding section can also be written like this:

```
phonenumber[position()=1 and @type='home']
```

The expressions are combined using the `and` operator. Only if both expressions are `true` does the entire expression return `true`. You also can create expressions that return `true` if only one of them is `true`. To do so, you use the `or` operator like this:

```
phonenumber[position()=1 or @type='home']
```

The expressions you use in predicates can be used with `xsl:if` and `xsl:when` as well. Because you cannot add two predicates together in that case, you have to use the Boolean operator `and` to combine expressions. Listing 6.19 shows how to use `xsl:choose` with an `xsl:when` element with such a `test` expression.

LISTING 6.19 Expression Using the Boolean Operator `or`

```
<xsl:choose>
  <xsl:when test="@type='fax'">
    Fax:   <xsl:value-of select="." />
  </xsl:when>
  <xsl:when test="@type='mobile' or @type='home'">
    Phone: <xsl:value-of select="." />
  </xsl:when>
</xsl:choose>
```

The output from applying Listing 6.19 to Listing 6.1 is shown in Listing 6.20.

OUTPUT **LISTING 6.20** Result from Applying Listing 6.19 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*

              Phone: 555-1234
              Fax:   555-2345
              Phone: 555-3456
      *Carol Waitress*

              Phone: 555-5432
              Phone: 555-9876
```

ANALYSIS In Listing 6.20, both the home and mobile phones are treated as regular phones. Line 5 in Listing 6.19 uses an expression combined using the `or` operator to accomplish this result. If either expression to the left or right of the operator is `true` (or both), the node is processed.

You can combine multiple expressions by using multiple `or` and `and` operators.

Caution The `and` operator has precedence over the `or` operator. This means that in the expression `A or B and C`, either B and C need to be `true` or A needs to be `true`. You can manipulate the order of precedence by using parentheses. In the expression `(A or B) and C`, the part between the parentheses is evaluated first and then used to evaluate the rest of the expression.

Using Boolean Functions

You've learned that expressions evaluate to Boolean values. These values, `true` and `false`, aren't available as literal values, but two corresponding functions represent these values: `true()` and `false()`. These functions aren't useful in most expressions because an expression already returns a Boolean value. Testing against them is therefore useless. Only in complex expressions combining functions and Boolean operators can these functions serve a purpose. They can be handy during the development process if you want to force the stylesheet to process certain code, so you can see whether it produces the right output when the code inside is invoked. For instance, if you have an `xsl:choose` element and want to force the `xsl:otherwise` code, you can temporarily set all the `test` attributes of the `xsl:when` elements to `<xsl:when test="false()">`. That way, the code inside the `xsl:when` elements is never invoked. Listing 6.21 shows such a sample.

LISTING 6.21 Using `false()` to Force Flow of Control

```
1:  <xsl:choose>
2:    <xsl:when test="false()">
3:      Fax:    <xsl:value-of select="." />
4:    </xsl:when>
5:    <xsl:when test=" false()">
6:      Mobile: <xsl:value-of select="." />
7:    </xsl:when>
8:    <xsl:otherwise>
9:      Phone:  <xsl:value-of select="." />
10:   </xsl:otherwise>
11: </xsl:choose>
```

Applying Listing 6.21 to Listing 6.1 yields the result in Listing 6.22.

OUTPUT **LISTING 6.22** Result from Applying Listing 6.21 to Listing 6.1

```
<?xml version="1.0" encoding="utf-8"?>
      *Joe Dishwasher*

              Phone:  555-1234
              Phone:  555-2345
              Phone:  555-3456
      *Carol Waitress*

              Phone:  555-5432
              Phone:  555-9876
```

ANALYSIS As you can see from the result in Listing 6.22, only the `xsl:otherwise` block on lines 8 through 10 in Listing 6.21 is executed. The `xsl:when` blocks are ignored because their `test` attribute contains the function `false()`.

Negating an Expression Result

You've learned that you can use different comparison operators so that you can do more than just test equality. If you want to test for inequality instead of equality, you can use the `!=` comparison operator instead. Another method is to use the `not()` function, which returns the opposite Boolean value from your expression. So, `not(A=B)` yields the same result as `A!=B`. Because you can use all these different operators, you may not see the value of this function. Basically, almost any comparison you can create by using `not()`, you also can create by using other operators. The `not()` function becomes useful after you take into account implicit conversion to Boolean values.

Conversion to Boolean Values

An expression doesn't necessarily return a Boolean value. An expression that doesn't compare anything but selects a node or node-set can return a number, string, node-set, or tree fragment. Because a predicate expression or an expression used in a `test` attribute expects a Boolean value, it implicitly converts the value to a Boolean value. Table 6.2 shows the rules used to convert the value to a Boolean value.

TABLE 6.2 Rules Applied When Converting Values to Boolean Values

Expression Result	*Conversion Rules*
Number	`false`: zero
	`true`: anything but zero
String	`false`: empty string
	`true`: nonempty string
Node-set	`false`: empty node-set
	`true`: nonempty node-set
Tree fragment	`false`: all text nodes empty
	`true`: at least one nonempty text node

In Table 6.2, the results for numbers and strings are probably clear to you. Node-sets and tree fragments are a little more difficult. When used as a test on Listing 6.1, the following expression would return false:

```
/employees/employee/kids
```

There are no `kids` elements, so this expression returns an empty node-set and thus `false`. The expression

```
/employees/employee/phonenumber
```

returns a nonempty node-set and thus `true`.

Tree fragments are even trickier. A tree fragment consisting of only empty nodes, possibly with attributes, returns `false`. So `/employees` would return `false` because no elements contain text. However, if you convert Listing 6.1 into a document with only elements, it returns `true` because at least one element contains text.

So much for implicit conversion, but what if you want to compare the Boolean values of two non-Boolean values? Because the implicit conversion takes place after the entire expression is evaluated, comparing these values is not possible without a little help. This help comes in the form of the `boolean()` function. The expression

```
boolean(employee[1]/phonenumber) and boolean(employee[2]/phonenumber)
```

implicates that both employees must have at least one phone number, or the whole deal is off. This is case because the `boolean()` function converts the node-sets passed as arguments (between the parentheses) to a Boolean value before combining the results of the separate functions with the `and` operator.

Summary

Today you learned that you can use `xsl:for-each` to select and iterate through a node-set. The advantage of using this approach over using templates is that no side effects occur because of nodes you didn't handle. Using templates as much as possible is recommended. They are the heart and soul of XSLT, which is all about data-driven development. Most processors are therefore optimized for templates as opposed to using `xsl:for-each`.

You also learned that you can use `xsl:if` and `xsl:choose` to branch off your flow of control so that certain code is executed only when certain conditions are satisfied. By using all kinds of comparison operators, you can create very complex conditions. You also can join multiple expressions by using multiple predicates or the Boolean operator `and`. With its brother `or`, you can also create expressions in which either of two conditions may be satisfied to execute certain code.

Finally, you learned that when an expression does not return a Boolean value, the value is implicitly converted to a Boolean value. You can force this conversion by using the `boolean()` function.

Now that you have learned about all the basic elements and functions in XSLT, you're ready to learn about creating different output types and manipulating whitespace behavior. That will be the subject of tomorrow's lesson.

Q&A

Q Is there any way that expressions I create will return an error?

A No. If an expression works—that is, if its syntax is correct—it will never return an error. Whether it does what you want it to do is another matter. XSLT has been created in such a way that errors are possible only when the syntax is incorrect, so when your stylesheet works, it will continue to work, no matter how crazy the input.

Q All those implicit functions make it hard to see what's going on. Why shouldn't I write out everything?

A Getting used to implicit functions (and conversion) takes some time. If you want to be certain that the right thing happens, explicitly writing all code is good practice. You also can write code using the shortcuts and write explicit code only when something doesn't work the way it should.

Q Can `xsl:when` exist outside an `xsl:choose` element?

A No. `xsl:when` may occur only within `xsl:choose`. `xsl:choose` also must contain at least one `xsl:when` element.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: `xsl:for-each` can be a child element of the `xsl:stylesheet` element.
2. True or False: `xsl:choose` must contain an `xsl:otherwise` element.
3. What is the difference between selecting data and matching data?
4. What is the difference between using multiple `xsl:if` elements after another and `xsl:when` elements with the same test expressions inside an `xsl:choose` element?
5. Is there a difference between expressions within a predicate and expressions used with the `test` attribute of the `xsl:if` and `xsl:when` elements?

Exercises

1. From the following XML, create a stylesheet that rearranges the document so that the cars are grouped by year. To do so, use `xsl:for-each` and predicate expressions.

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

2. Create a stylesheet for the XML from Exercise 1 to add an attribute named `country` to each element. For Ford and Chevrolet, this attribute should read `'USA'`; for Honda and Toyota, `'Japan'`; and for Volkswagen, `'Germany'`. Use either `xsl:if` or `xsl:choose`.

DAY 7

Controlling the Output

In yesterday's lesson, you learned how to select and operate on data by using pull processing, which is the opposite of push processing employed when nodes are matched by templates. This capability allows you to manipulate the flow of control in the stylesheet. You also learned about conditional processing, so code is executed only if certain conditions are met. With this knowledge, you can insert elements, attributes, and text at the point in the output where you want it. This, however, doesn't give you 100% control over the output because the output is still rendered with the default output parameters.

The focus of today's lesson is manipulating these parameters so that every element, attribute, or piece of text is precisely where you want it. You need this tight control over the method used to render the output when you want to create documents other than XML. You also need to be able to control whitespace and encoding, which also are covered in today's lesson.

Today you will learn how to do the following:

- Create different forms of output, such as HTML
- Create output in specific character encodings
- Work with text that is not output escaped
- Manipulate the whitespace in the output stream

Creating Different Output Formats

One great thing about XSLT is that you don't have to create XML output. You can actually create many (text-based) output formats. You need to create valid XML only when you want to use the resulting document with an XML parser or an XSLT processor. To assist you in creating different types of documents, XSLT provides three default output formats from which you can choose: XML, HTML, and plain text.

When the processor creates output, it actually creates an internal representation of the output and then outputs that representation by using the chosen output method. The actual output rendered also depends on parameters that are relevant for the chosen output method.

Understanding XML Output

By default, the XSLT processor creates XML output, which is what you did in previous lessons. That does not mean, however, that all output is well formed XML. To start with, well-formed XML needs to have a root element. Remember that some of the stylesheets from earlier lessons created output that didn't have a root element. The stylesheets that created documents with a root element produced well-formed XML. The others produced an XML declaration, followed by text that was properly output escaped.

New Term Basically, when the processor is supposed to output XML, the output needs to be either well-formed XML or an external general parsed entity, which is not exactly XML. An *external general parsed entity* is a piece of text that might contain an XML declaration and content. This content can be text, XML elements with or without attributes, character entities, and so on, in any sequence. Basically, any content is valid for an external general parsed entity if it would be valid as the value of an element in an XML document.

The following are some examples of external general parsed entities:

```
Hello World!
<?xml version="1.0" encoding="utf-8"?>Hello World!
<text>Hello World!</text>
<text>Hello</text><text>Hello World!</text>
This is a &quot;<greeting>Hello</greeting> World!&quot; <type>sample</type>
```

Although the preceding examples are not well-formed XML documents, they could be part of an XML document. Well formed XML and external general parsed entities are different things, but they overlap in some areas. So, in some cases, an XML document is an external general parsed entity, and vice versa. Well-formed XML documents containing document type declarations are not external general parsed entities.

Note

In most applications, the difference between well-formed XML and external general parsed entities is not very important. In most cases, you will probably work with the former.

If the output generated is neither well-formed XML nor an external general parsed entity, the output is basically in error. However, because the XSLT specification isn't very clear about what should happen if that situation occurs, most processors just generate the output. They might also report an error, but that is not required.

Note

Even if the output is well-formed XML, it is by no means required to be valid XML—that is, XML validated by a Document Type Definition (DTD) or Schema. Whether or not the output is well-formed or valid XML is interesting only for the application consuming the output.

Forcing XML Output

The default output method of the processor is XML, which generates output as discussed in the preceding sections. If the root element of the output being generated is `<html>`, however, the processor might choose to use the HTML output method instead. Although currently the XSLT specification has support only for these two SGML derivative languages, support for other methods might be added as XSLT evolves. Also, processors can implement extensions that allow for other output methods as well.

NEW TERM If you want to be sure that the generated output will be XML, you can use the `xsl:output` element, which is a top-level element and might occur several times in a stylesheet. A *top-level* element might occur only as a child element of the `xsl:stylesheet` element, not as a child element of any other element.

If multiple `xsl:output` elements exist in the stylesheet, all their attributes are merged, and the result is used as though there is only one `xsl:output` element. If attributes are conflicting, the processor might report an error or use the attribute that occurs last in the document.

To force the processor to output XML, you just need to add the following to a stylesheet:

```
<xsl:output method="xml" />
```

Note You can place the `xsl:output` element anywhere in the document, as long as it is a direct child of the `xsl:stylesheet` element. It doesn't matter if that element is at the beginning, middle, or end of the document. The convention is to insert it before any `xsl:template` elements.

Apart from forcing the output to be XML, the preceding code does nothing. The stylesheet's output would therefore not be any different if you left it out. So, why bother putting it in? First, you do so to show any stylesheet readers, human or otherwise, that the stylesheet will generate XML. Furthermore, the `xsl:output` element has more attributes, some of which influence what the output will look like. Because not all attributes are relevant at this point, not all attributes will be discussed in this lesson.

Note In Appendix B, "XSLT Reference,"you can find a list of all the attributes of each element.

Specifying the XML Version of the Output

Currently, there is only one version of XML: version 1.0. In light of this fact, specifying a version explicitly might seem foolish. However, if a new version were to come into existence, your resulting document might be output in that version, possibly making it useless with older parsers and processors. So, like explicitly specifying that you are generating XML output, always explicitly specifying the version is wise, just so that nothing is left to chance and you end up with an application that breaks. When you specify the version, the `xsl:output` element looks like this:

```
<xsl:output method="xml" version="1.0" />
```

Because only XML version 1.0 is in use at this point, the value of the `version` attribute should always be 1.0. If you specify a different version, the output differs per processor. Saxon, for instance, creates an XML declaration containing the given version, whereas MSXML ignores it and outputs an XML declaration with version 1.0 instead. Some processors might choose to report an error.

Listing 7.1 shows what an entire stylesheet typically looks like when it forces XML output of version 1.0.

LISTING 7.1 Stylesheet Using `xsl:output`

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <xsl:stylesheet version="1.0"
3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
4:
5:    <xsl:output method="xml" version="1.0" />
6:
7:    <xsl:template match="/">
8:      <xsl:copy-of select="." />
9:    </xsl:template>
10: </xsl:stylesheet>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS The code in Listing 7.1 only copies the source document to the output, adding an XML declaration if it wasn't already there or changing it if it was different. The `xsl:output` element on line 5 forces XML output with version 1.0.

Removing the XML Declaration from the Output

The XML declaration that most processors put on top of every XML document created is not mandatory. If you want, you can tell the processor to leave the declaration out (or force it in). The following code, apart from defining the output method and version, removes the XML declaration from the output:

```
<xsl:output method="xml" version="1.0" omit-xml-declaration="yes" />
```

The `omit-xml-declaration` attribute more or less speaks for itself. It can have the value `yes` or `no`. In the latter case, the declaration is forced in; otherwise, it is forced out.

Note If the XML declaration is not written, the `method` and `version` attribute might still be used internally by the processor.

Creating HTML Output

Because HTML is not well formed like XML, processors have a separate output method to create HTML. This method can be invoked explicitly with `xsl:output`, but some processors will automatically create HTML output if no output method is specified and the first element in the output is `<html>`. Because this is not required, you should specify that you want to generate HTML output. If you want to generate HTML output, the `xsl:output` element should look like this:

```
<xsl:output method="html" />
```

Caution

Unlike HTML, XHTML is well formed. To create correct XHTML output, you need to use the XML output method explicitly. With the HTML output method, the processor generates tags that are not well formed for elements such as `<br>`, `<hr>`, and `<input>`. These tags are not allowed in XHTML.

There are some major differences between XML output and HTML output. The most obvious is that XML is well formed and HTML is not. This difference manifests itself in elements that do not have closing tags. They are similar to empty XML tags, but without the ending slash, such as `<emptytag />`. The following HTML 4.0 elements do not have closing tags:

`<area>`	`<base>`	`<basefont>`
`<br>`	`<col>`	`<frame>`
`<hr>`	`<img>`	`<input>`
`<isindex>`	`<link>`	`<meta>`
`<param>`		

Another difference is that HTML has some empty attributes, such as `<option selected>`. In XML and XHTML, using empty attributes is not allowed, so you must write this example as `<option selected="selected">`. If you create an element like this in a stylesheet, the HTML output method sees to it that this element is written to the output as `<option selected>`.

Furthermore, in HTML you can have `script` or `style` elements. Within these elements, characters such as `<`, `>`, and `&` do not need to be output escaped. If you output escape them, the code in these elements fails to function properly in HTML. When you use the HTML output method, the output escaping in these elements is removed.

A final difference between XML and HTML is the notation of processing instructions. In XML, processing instructions are in the form `<? instruction ?>`, whereas in HTML they are in the form `<? instruction >`. You can see that the terminator is different. When HTML output is generated, the processing instruction is written to the output in the HTML form.

Note

Processing instructions aren't used often in HTML. They are discussed here to provide you with a complete picture.

One often forgotten point is that XML is case sensitive, and HTML is not. In HTML, the code `<b>bold</B>` is perfectly legal, whereas in XML it is not. Fortunately, case is not a problem when you go from XML to HTML. The XML and stylesheet are case sensitive, and the HTML output follows the case used in the stylesheet.

Specifying the HTML Version of the Output

Several versions of HTML are in use. The most commonly used versions are 3.2 and 4.0. If you use the HTML output method, the default value is 4.0. This value is used only by the processor to determine what the output should be and does not show up in the output in any shape or form. If you want to specifically output HTML 3.2, the `xsl:output` element should look like this:

```
<xsl:output method="html" version="3.2" />
```

HTML Creation Sample

The best way to show that the HTML output really works is by example. Listing 7.2 shows an XML document used in earlier lessons; here, it will be transformed to HTML.

LISTING 7.2 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

ANALYSIS Listing 7.2 does not contain any text nodes, just attribute values. No whitespace will be copied from the source to the output. Listing 7.3 shows the stylesheet used to create the HTML.

LISTING 7.3 Stylesheet Creating HTML from Listing 7.2

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" />

  <xsl:template match="/">
    <form method="POST" action="post.asp">
      <xsl:for-each select="/cars/car">
        <INPUT type="radio" name="car" value="{@model}" />
```

LISTING 7.3 Continued

```
11:         <xsl:value-of select="@manufacturer" />
12:         <xsl:value-of select="@model" /><br />
13:       </xsl:for-each>
14:     </form>
15:   </xsl:template>
16: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 7.3 creates an HTML form and for each element an option button. The latter is done with the `<INPUT>` element on line 10. The case of the elements created, as well as the attributes, is mixed. This example just shows that the processor will not touch the case of elements and attributes when generating HTML output. Note that the `<INPUT>` is well formed, as it should be in XML (and thus XSLT). This also goes for the `<br />` element. Line 10 uses the syntax with the curly braces to give a value dynamically to the `value` attribute of the `INPUT` element. The expression inserts the value of the `model` attribute of the current node. Listing 7.4 shows the result if you apply Listing 7.3 to Listing 7.2.

OUTPUT **LISTING 7.4** Result from Applying Listing 7.3 to Listing 7.2

```
<form method="POST" action="post.asp">
➥<INPUT type="radio" name="car" value="Focus">FordFocus<br>
➥<INPUT type="radio" name="car" value="Golf">VolkswagenGolf<br>
➥<INPUT type="radio" name="car" value="Camry">ToyotaCamry<br>
➥<INPUT type="radio" name="car" value="Civic">HondaCivic<br>
➥<INPUT type="radio" name="car" value="Prizm">ChevroletPrizm<br>
➥</form>
```

ANALYSIS In Listing 7.4, the processor does not output the XML declaration. It shouldn't because the output is not XML and also not well formed. The case used in the stylesheet is retained. Also, the `<INPUT>` and `<br>` elements are output as proper HTML, without the closing tag or closing slash for empty elements. Compare the result in Listing 7.4 with the result in Listing 7.5.

OUTPUT **LISTING 7.5** Result Not Using `xsl:output`

```
<?xml version="1.0" encoding="utf-8"?>
<form method="POST" action="post.asp">
<INPUT type="radio" name="car" value="Focus"/>FordFocus<br/>
<INPUT type="radio" name="car" value="Golf"/>VolkswagenGolf<br/>
<INPUT type="radio" name="car" value="Camry"/>ToyotaCamry<br/>
<INPUT type="radio" name="car" value="Civic"/>HondaCivic<br/>
<INPUT type="radio" name="car" value="Prizm"/>ChevroletPrizm<br/>
</form>
```

ANALYSIS Listing 7.5 shows the output from the same stylesheet as Listing 7.4, but without the `xsl:output` element specifying that HTML output should be generated. The output differs in some key areas: The XML declaration is added, and all elements are well formed.

Creating Other Types of Output

The last output method is the text output method, which is used to create output other than HTML and XML. You can use it to create text with other formatting styles, such as PostScript or text-based formatting used in older desktop publishing systems. You also can create delimited data files, such as comma-delimited files, application source code, or character- or text-based image formats.

If you use the text method, you can still create elements with attributes as you would normally. The catch is that all element tags, including any attributes, are stripped before the output is rendered. Also, any output-escaped characters are written to the output as the original characters.

Listing 7.6 shows the result if you change line 5 of the stylesheet in Listing 7.3 into `<xsl:output method="text" />` and apply it to Listing 7.2.

OUTPUT **LISTING 7.6** Output from a Stylesheet with the Text Output Method

```
FordFocusVolkswagenGolfToyotaCamryHondaCivicChevroletPrizm
```

ANALYSIS There is not much to the output in Listing 7.6. It is exactly the same as Listings 7.4 and 7.5, but with all elements, attributes, and declarations removed. Because the other outputs didn't have whitespace, there is none in this one either, but this lack of whitespace is more apparent because of the lack of anything else.

Binary Data

The major drawback of XSLT is that the current version can't create documents in some sort of binary encoding without help from another program or software component. This means that with just a processor you can't create formats such as Portable Document Format (PDF) or some common formats for word processors. Most word processors offer some text-based exchange format that you can use, but they often result in output that is not exactly the same as when that document is native to the word processor.

It is a misperception to think that XML with XSLT is going to replace these formats any time soon. Formats such as PDF and Microsoft Word documents are in wide use and cannot be changed quickly. The companies that created these formats are also not likely to give up on them so easily. Formats such as PDF are also used as formats for electronic books. These formats can include digital signatures and such to make the files readable only to some people or for a fixed number of times so that the files can be sold. Without a means to do similar things with XML documents, these formats will continue to exist. This means that it is likely that at one point or another, you will want to create these formats from existing XML documents.

XML documents can contain Base64-encoded data elements, but XSLT can't create elements that are Base64 encoded from data elements and attributes that are not. The fact that this isn't possible with just XSLT doesn't come as a surprise. Many binary formats are proprietary. What would you say if XSLT incorporated conversion to a certain binary format and not for another? In that case, the W3 Consortium (W3C) would no longer be seen as an independent organization that produces standards but would be seen as partial to certain formats.

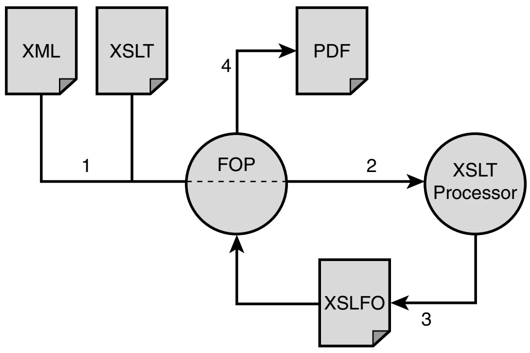
The fact that XSLT itself cannot create binary data formats does not mean that doing so is entirely impossible. One of the most common scenarios for organizations publishing documents is storing documents in XML and generating output in different formats. Currently, tools exist to create Portable Document Format (PDF) and Rich Text Format (RTF) documents from XML. These tools use XSL (XSLT + XSLFO) and a conversion tool. The conversion tools convert the output from the processor (which is XML) to either PDF or RTF.

The Apache XML Project is working on a tool that can be used to create PDF, Printer Control Language (PCL), and other formats. At the time of this writing, this tool, called FOP, is still very much under development, but you can download it from `http://xml.apache.org/fop/index.html`.

FOP can be called from the command line like a processor. You just have to specify XML and XSLT documents to transform the XML document into a document formatted using XSLFO. Under the covers, FOP invokes an XSLT processor that creates the XSLFO-formatted file, which FOP then converts to PDF. This process is schematically depicted in Figure 7.1.

FIGURE 7.1

Diagram of XML-to-PDF conversion.



The process depicted in Figure 7.1 consists of four steps. In step 1, FOP is called and is provided with the XML and XSLT documents needed. In step 2, FOP invokes the XSLT processor, passing on the provided XML and XSLT. In step 3, the processor transforms the XML with the given XSLT, which results in an XSLFO-formatted document that is passed back to FOP. In the final step, FOP converts the XSLFO-formatted document and outputs it as a PDF file.

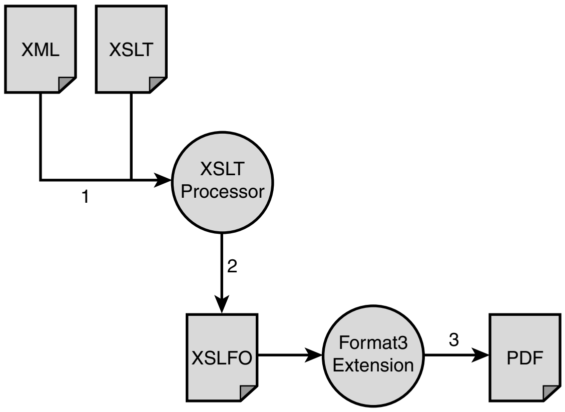
A tool similar to FOR is the Java xsl-FO to Rtf converter (jfor). This tool, which you can download from `http://www.jfor.org`, can be used to create RTF documents from the same documents that FOP uses to create PDF documents.

Processor Extensions for Other Output Formats

XSLT has been designed so that it can be extended with functionality that is not part of the XSLT specification. Most processors have some kind of mechanism to extend them. The added functionality can be used from XSLT to create output that is normally not possible. It is very likely that existing processors will be extended to accommodate transformation processes like XML-to-PDF conversion as discussed in the preceding section. Using processor extensions turns this process upside down. Instead of an application calling the processor to transform XML, the processor calls the extension to do something based on code in the stylesheet. This approach is similar to a browser that uses plug-ins to show you certain types of files. The process in Figure 7.1 is shown using this approach in Figure 7.2.

FIGURE 7.2

Diagram of XML-to-PDF conversion using XSLT extensions.



The process in Figure 7.2 consists of only three steps. In step 1, the processor is called with the XML and XSLT input. In step 2, the processor transforms the XML and passes that output to the extension. In step 3, the extension converts the output from the processor to PDF. This process could be triggered by an `xsl:output` element that looks like this:

```
<xsl:output method="format:pdf" />
```

Note that the `pdf` format is prefixed by a namespace. This is necessary to tell the processor that it is dealing with an extension.

Note

On Day 20, "Working with XSLT Extensions," you will learn more about processor extensions.

Specifying the Media Type

The `xsl:output` element has an attribute named `media-type`. This attribute has been added so that you can specify the media type (or MIME type) of the resulting output. The XSLT specification does not specify how this information is to be used, however, so most processors ignore it. Using it certainly does not affect processing, but the media type might be used when the output is stored or transmitted. The media type information might be passed on to the environment in which the processor is used. It might, for instance, be passed in to a Web server and added to the HTTP protocol header. If it is not specified, the media type assumed is `text/xml`.

Suppose your stylesheet generates a new XSLT stylesheet, which you want to give the media type `application/xml` instead of `text/xml`. In this case, the `xsl:output` element would look something like this:

```
<xsl:output method="xml" version="1.0" media-type="application/xml" />
```

A stylesheet that creates a JavaScript file with the text output method might contain the following:

```
<xsl:output method="text" media-type="application/x-javascript" />
```

Output Encoding and Output Escaping

On Day 2, "Transforming Your First XML," you learned about the different encoding types you can use with XML—most notably UTF-8 and UTF-16—and also some ISO encoding types. Also, on Day 5, "Inserting Text and Elements," you learned that special characters need to be output escaped to work properly in XML. To create different forms of output, you need to be able to control which encoding is used and what happens with output-escaped characters.

Determining the Output Encoding

Until now, you have had no control over the encoding of the output. The processor chose its default output encoding and was done with it. Because not all applications support all encoding types, you might need to create output that uses a specific encoding type. For most XML applications, this doesn't apply, but when you're creating text output meant for some application, it most certainly will. UTF-16 encoding, for instance, is releatively new. Older applications can't deal with text formatted in UTF-16, so if you want to create output meant for these applications, you need to create a document with an encoding type that is supported, such as ASCII or UTF-8.

The processors discussed previously all use UTF-8 as default encoding, except for MSXML, which uses UTF-16 encoding. A processor should support either UTF-8 or UTF-16, but it doesn't have to support both. When it does, the processor determines which it uses as the default.

Providing a processor supports a certain encoding type, you can tell the processor to use that encoding type when creating the output. The `xsl:output` element contains the `encoding` attribute to do so. If you want to force UTF-8 encoding, the following would work:

```
<xsl:output encoding="UTF-8" />
```

The `encoding` attribute can be used with any of the output methods and can have a value indicating any of the valid encoding types for XML. These encoding types were discussed on Day 2, but just to remind you, they are as follows:

- Unicode: UTF-8 or UTF-16
- ISO/IEC 10646: ISO-10646-UCS-2 or ISO-10646-UCS-4
- ISO 8859: ISO-8859-1, … ISO-8859-*n* (*n* is the part number)
- JIS X-0208-1997: ISO-2022-JP, Shift_JIS, or EUC-JP

Processors should always support UTF-8 or UTF-16, but supporting additional types isn't mandatory. Many processors do not support all encoding types. If the processor doesn't support the defined encoding type, it ignores the value of the `encoding` attribute and creates output in the default encoding. In that case, the processor might report an error, telling you that it doesn't support the requested encoding and that it will use another.

With the XML output method, the encoding type of the output is stated in the XML declaration, which was discussed on Day 2. The following example declaration tells you that the output is encoded in ISO-8859-1 encoding:

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

Encoding Characters That Are Not Supported

The different encoding types support different character sets. Some encoding types have a much broader character base than others. UTF-16 encoding, for instance, can theoretically accommodate 65,536 characters, whereas UTF-8 can accommodate only 256 characters. You can easily see that UTF-16 probably has many characters that don't exist in

UTF-8. If you create UTF-8 output from a UTF-16 source document, you are faced with the question of what happens to those characters that can't be represented in UTF-8.

If you're creating text output, you cannot circumvent the problems arising from characters that are not supported. Because the output being created is in no way related to XML and could be a format that has very specific rules, you can't use output escaping, for example, to get around the deficiencies of the output encoding. Even if you could use output escaping, you would have no guarantee that all applications to which the output is aimed will be able to properly understand the output escaping used. This is why the processor reports an error for any character that it can't represent properly in the chosen output encoding. For example, Kanji characters cannot be automatically converted to Greek.

For XML (including XSLT) or HTML output, there is a way out. Any characters that cannot be represented with the chosen output encoding are output escaped according to the rules of XML and HTML. Because parsers and processors treat characters and output-escaped characters as semantically the same, the output document is correct regardless of the encoding.

Disabling Output Escaping

NEW TERM In some cases, being able to write text as is into XML, without having to use output escaping, might be useful. This capability is particularly useful if an element is supposed to contain code such as C, Java, or Visual Basic because code often needs to be output exactly as it appears in the source if it is going to function properly. For this reason, XML might contain *CDATA sections,* which are elements that contain character data that is not output escaped. Even though characters in CDATA sections are not output escaped, they do not interfere with processing.

CDATA sections start with a `<![CDATA[` delimiter and end with `]]>`. To see the significance of CDATA sections, look at this line of code:

```
<compare>X < Y</compare>
```

Because `X < Y` contains a character that is used by XML tags, this line of code produces an error when loaded by a parser because it is not well-formed XML. The following line of code is well formed, however, and does not generate an error:

```
<compare><![CDATA[X < Y]]></compare>
```

Although this example looks very different from a normal element value, it isn't that different. In fact, when the parser loads this value, its internal representation is the same as for the following code:

```
<compare>X &lt; Y</compare>
```

This fact can be best demonstrated with a sample. Listing 7.7 copies the entire source XML document to the output.

LISTING 7.7 Stylesheet Copying the Source to the Output

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

If you feed the stylesheet in Listing 7.7 the XML shown in Listing 7.8, the result is as shown in Listing 7.9.

LISTING 7.8 XML with CDATA Sections

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <operators>
3:    <compare><![CDATA[X < Y]]></compare>
4:    <compare>X &gt; Y</compare>
5:    <concat><![CDATA[X & Y]]></concat>
6:  </operators>
```

OUTPUT **LISTING 7.9** Result from Applying Listing 7.7 to Listing 7.8

```
<?xml version="1.0" encoding="utf-8"?><operators>
  <compare>X &lt; Y</compare>
  <compare>X &gt; Y</compare>
  <concat>X &amp; Y</concat>
</operators>
```

ANALYSIS In Listing 7.8, lines 3 and 5 use CDATA sections, whereas line 4 uses output escaping. In Listing 7.9, these lines are all output as though they are the same. In the output, each line no longer has a CDATA section and is output escaped instead. This example effectively shows that the internal representation for the different elements is the same.

If you're creating XML output, you can define which elements need to be output as CDATA sections by using the `cdata-section-elements` attribute part of the `xsl:output` element. This attribute's value should be a whitespace-separated list with the names of elements that need to be output as CDATA sections. So, if you want to output all `compare` elements in Listing 7.8 as CDATA sections, you should add the following line to Listing 7.7 as a top-level element:

```
<xsl:output cdata-section-elements="compare" />
```

Listing 7.10 shows the result when this line is added to Listing 7.7.

OUTPUT **LISTING 7.10** Result of Applying Listing 7.7's Stylesheet with CDATA Sections

```
<?xml version="1.0" encoding="utf-8"?><operators>
  <compare><![CDATA[X < Y]]></compare>
  <compare><![CDATA[X > Y]]></compare>
  <concat>X &amp; Y</concat>
</operators>
```

ANALYSIS As you can see in the result in Listing 7.10, adding the `cdata-section-elements` attribute to Listing 7.7 forces the `compare` elements to be output as CDATA sections. The `concat` element, which is not specified in the `cdata-section-elements` attribute, is output just like before usingoutput escaping.

The processor is not required to output CDATA sections, even if you tell it to. In some cases, it might not output these sections or might split them into several sections. This result is likely to occur when the output contains characters that are not available in the used output encoding, in which case output escaping is needed to output those characters correctly. CDATA sections cannot contain output-escaped characters because CDATA sections themselves are a form of output escaping.

Note

You might think that CDATA sections can be useful for storing binary data formats. However, because the end delimiter of CDATA sections `]]>` might occur in such data, you should use Base64 encoding or something similar to store such data to prevent the occurrence of these characters. When you do that, using a CDATA section is no longer necessary because Base64 encoding effectively removes the characters for which you used the CDATA section in the first place. Using CDATA sections makes sense, however, when you need to add XML, XSLT, or HTML code in a document without it being handled by the parser or processor as such.

CDATA sections are important, for example, when you use XML and XSLT to create HTML documents with JavaScript code. JavaScript code often contains characters not allowed, as is in XML, and whitespace, specifically line feeds, is significant. In a

CDATA section, you can insert the code exactly as it should appear in the output, which is much easier than getting it right as proper XML. One problem is that when you create HTML, the CDATA section is removed and the text output escaped. This renders the JavaScript code useless. How you can solve this problem is discussed later in this lesson.

Controlling Whitespace

Handling whitespace in XSLT and XML is often a source of frustration. Whitespace can pop up in your output where you don't want it, or it might be removed where you want it left in. If you want to control when and where whitespace is inserted or removed, you need to know how parsers and processors deal with it.

Understanding Whitespace

XML contains several characters that are treated as whitespace. Other characters, although not visible, are not treated as whitespace by parsers and processors. These characters often carry information that has significance and might therefore not be removed. One such character is the nonbreaking space character familiar in HTML. In XML, this character is represented using the character reference `&#160;` or `&#xA0;` and is treated as a nonwhitespace character. It is not treated as a whitespace character because it is specifically inserted whitespace with a very specific function. As far as XML is concerned, a space (`&#x20;`), tab (`&#x9;`), carriage return (`&#xD;`), and new line (`&#xA;`) are whitespace characters. This does not mean that these characters never have significance. XML makes a distinction between significant and insignificant whitespace based on the location of whitespace in an XML document. Listing 7.11 helps you to understand the difference.

LISTING 7.11 XML Document with Whitespace

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <news>
3:    <header>Sprinter John Benson runs 100 metres in under 5 seconds</header>
4:    <date>09/12/2040</date>
5:    <text>
6:      In a phenomenal race against rivals Lewis Carlson and Chris Linford,
7:      Canadian sprinter John Benson ran the 100 meters in a new world record
8:      time of 4.98 seconds.
9:    </text>
10: </news>
```

ANALYSIS Listing 7.11 contains several sections of whitespace. Each child element of the `news` element is preceded by several whitespace characters, and whitespace appears between the closing tags of the `text` and `news` elements on lines 9 and 10. Technically, this whitespace is part of the value of the `news` element, but because the news element contains only child elements and no text, the whitespace isn't very significant. If you remove the indentation of the header and date elements on lines 3 and 4, the data in the document is not affected. Even if you put these two elements on the same line, the value is still the same. The whitespace in the value of the text element is a different matter, however. If you remove the newline characters and indentation, the value changes. This change is probably not significant to a human reader but is very significant to a nonhuman reader. This is exactly the problem: For some whitespace, a processor or parser cannot determine whether it can be removed without having an impact on the information.

When a parser creates a representation for an XML document, whitespace in between elements of the document are represented as nodes containing only whitespace. Whitespace that is part of a text value, on the other hand, remains part of that value and is not represented by a whitespace node. When this representation is passed on to the processor, the processor determines how to deal with whitespace-only nodes.

Stripping Whitespace from the Source Document

By default, XSLT leaves in place all whitespace in the source document. If you create a copy of a document or a tree fragment using a stylesheet like the one in Listing 7.7, you end up with a copy that includes all original whitespace, including whitespace-only nodes. Getting rid of these whitespace-only nodes is fortunately quite simple when you use the xsl:strip-space element. This top-level element might occur only as a child element of the xsl:stylesheet element. The value of the elements attribute of the xsl:strip-space element, which is the only attribute, must contain a whitespace-separated list of element names. Any whitespace-only node from the source document that is a child of either of the named elements will be stripped. Listing 7.12, which is a variation on Listing 7.7, demonstrates this use.

LISTING 7.12 Stylesheet Stripping Whitespace-Only Element

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:strip-space elements="news" />

  <xsl:template match="/">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

If Listing 7.12 is applied to Listing 7.11, the result is similar to Listing 7.13.

OUTPUT **LISTING 7.13** Result from Applying Listing 7.12 to Listing 7.11

```
1:  <?xml version="1.0" encoding="utf-8"?><news><header>Sprinter John Benson
    ➥  runs 100 metres in under 5 seconds</header><date>09/12/2040</date>
    ➥<text>
2:      In a phenomenal race against rivals Lewis Carlson and Chris Linford,
3:      Canadian sprinter John Benson ran the 100 meters in a new world record
4:      time of 4.98 seconds.
5:    </text></news>
```

ANALYSIS In Listing 7.12, an `xsl:strip-space` element appears on line 5. This element has a profound impact on what the copy of Listing 7.11 looks like. Without it, the copy would have been identical, but not all whitespace-only nodes from the source document have been removed. The result is a very long line 1 in Listing 7.13, containing nearly all elements separated by nothing at all. The code continuation markers show just how long line 1 is. Now look at the value of the `text` element. There, the whitespace is preserved and is identical to that of Listing 7.11. The element starts on line 1, where its first character is a new line. Notice that the indentation of the `text` element's closing tag is actually mistaken for whitespace that is part of the value and is thus inserted on line 5.

A major advantage of using the `xsl:strip-space` element is that you can specify multiple elements. In fact, you also can use wildcards to specify the elements from which you want to strip the whitespace-only nodes. So, the line

```
<xsl:strip-space elements="para news:*" />
```

removes the child whitespace-only nodes from the `para` element and all elements in the `news` namespace. This way, you can remove unwanted space easily and quickly.

Preserving Whitespace

When you use `xsl:strip-space` with wildcards, in some situations whitespace might be stripped from certain elements in which you actually want to preserve it. This is specifically the case when you're working with elements containing mixed content—that is, elements and text—such as in Listing 7.14.

LISTING 7.14 XML Element with Mixed Content

```
<text>
  Canadian <athlete>sprinter</athlete> <person>John Benson</person>
  ran the <event>100 meters</event> in a new world record time of
  <time>4.98 seconds</time>.
</text>
```

Analysis In Listing 7.14, the whitespace on line 2 between the `athlete` and `person` elements needs to be preserved. That whitespace is seen as a whitespace-only node and will therefore be stripped if the `text` element is specified in an `xsl:strip-space` element. If the text element were part of a document with many different elements and you wanted to strip the whitespace-only text nodes from all of them, you would use a wildcard to specify all elements. Unfortunately, that would also include the `text` element, so any whitespace elements in it would be removed although doing so might "damage" the value. With your current knowledge, that means you need to explicitly specify all elements that need whitespace stripped. The problem is that, at design time, it is likely that you don't know all elements beforehand. What you would like is to tell the processor to strip whitespace from all elements except the `text` element. Fortunately, you can do so by combining the `xsl:strip-space` element with the `xsl:preserve-space` element. The `xsl:preserve-space` element counteracts the `xsl:strip-space` element in that you can specify which elements should not be stripped of whitespace, even if they are specified by the `xsl:strip-space` element. Listing 7.15 helps to clarify this use.

Listing 7.15 Stylesheet Preserving Some Whitespace

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:strip-space elements="*" />
6:    <xsl:preserve-space elements="text" />
7:
8:    <xsl:template match="/">
9:      <xsl:copy-of select="." />
10:    </xsl:template>
11:  </xsl:stylesheet>
```

Analysis Any XML document to which the stylesheet in Listing 7.15 is applied will be copied with all whitespace stripped. That happens because, on line 5, the `xsl:strip-space` element specifies all elements with a wildcard. The only element that will not be stripped of whitespace-only nodes is the `text` element because the `xsl:preserve-space` element overrides the `xsl:strip-space` element for the `text` element.

Whether an `xsl:preserve-space` element overrides an `xsl:strip-space` element is determined by a calculated priority. As with the calculated priority of templates, the general rule is the more specific the selection, the higher its priority. Table 7.1 shows the calculated priorities of different selections.

TABLE 7.1 Calculated Priorities for Whitespace Stripping

Selection	*Priority*
Specific element (for instance, `model` or `car:model`)	0
Any element from a certain namespace (for instance, `car:*`)	–0.25
Any element (wildcard: `*`)	–0.5

As you can see in Table 7.1, a specific element has the highest priority. This means that you can easily use `xsl:strip-space` for a general case and then override it with `xsl:preserve-space` for specific cases.

`xsl:strip-space` and `xsl:preserve-space` elements are ignored for elements in the source document that contain an `xml:space` attribute. So, if you change the code in Listing 7.14 to match Listing 7.16, the whitespace between the `athlete` and `person` elements on line 1 is retained, regardless of any whitespace handling in a stylesheet.

LISTING 7.16 XML Element with Mixed Content

```
1: <text xml:space="preserve">
2:   Canadian <athlete>sprinter</athlete> <person>John Benson</person>
3:   ran the <event>100 metres</event> in a new world record time of
4:   <time>4.98 seconds</time>.
5: </text>
```

The `xml:space` attribute also can have the value `default`, in which case the definitions in the stylesheet come into play again.

Removing Whitespace in Values

In the preceding section, you learned how to strip whitespace-only nodes from a document. That leaves another type of very annoying whitespace—whitespace in a value, like that in Listing 7.13. In many cases, this whitespace is just there because of readability, indentation, or formatting by an editor. In any case, it is often a source of unwanted results because it can contain new lines, carriage returns, tabs, or multiple spaces after another. There is no way to get rid of this whitespace using XSLT elements. XSLT does, however, offer the `normalize-space` function, which you can use in select expressions. The `normalize-space` function removes all extra spaces and converts line feed, carriage return, and tab characters into single spaces. The resulting value has single spaces between words and other characters. Using the `normalize-space` function, you can re-create the code in Listing 7.13, but without the whitespace in the text element. The stylesheet shown in Listing 7.17 does just that.

LISTING 7.17 Stylesheet Using the `normalize-space` Function

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template match="*">
6:      <xsl:copy>
7:        <xsl:apply-templates />
8:      </xsl:copy>
9:    </xsl:template>
10:
11:   <xsl:template match="text()">
12:     <xsl:value-of select="normalize-space(.)" />
13:   </xsl:template>
14: </xsl:stylesheet>
```

ANALYSIS Instead of using `xsl:copy-of` for a deep copy, as in Listing 7.12, Listing 7.17 uses a "manual" deep copy. Each element it encounters is matched by the template on line 5, which calls `xsl:apply-templates` to invoke the same template for any child elements. If a text node is encountered, the template on line 11 is invoked. That template writes the text value to the output, but not before normalizing it, removing all redundant whitespace. The result of applying Listing 7.17 to Listing 7.11 is shown in Listing 7.18.

OUTPUT **LISTING 7.18** Result from Applying Listing 7.17 to Listing 7.11

```
<?xml version="1.0" encoding="utf-8"?><news><header>Sprinter John Benson
➥runs 100 meters in under 5 seconds</header><date>09/12/2040</date>
➥<text>In a phenomenal race against rivals Lewis Carlson and Chris
➥Linford, Canadian sprinter John Benson ran the 100 metres in a new
➥world record time of 4.98 seconds.</text></news>
```

ANALYSIS In Listing 7.18, all redundant whitespace is removed from the output. The output is effectively one line with no line feeds and so on. From a readability standpoint, this result is about as bad as it can get, but for data storage and manipulation, it is great because absolutely no side effects can get in your way when you deal with this document.

Whitespace Stripping Pitfalls

Stripping whitespace is tricky business. The last thing you want to do is remove whitespace that is significant—especially when you're working with elements containing both text and elements, such as tagged text. You need to be fully aware of the dangers you face when you use `xsl:strip-space` and `normalize-space`. The `normalize-space` function is best used on text-only nodes, not on mixed content. This way, you can ensure that you don't remove any significant whitespace between elements.

It is helpful to design your XML documents in such a way that you know which elements can contain mixed content and which cannot. From those that don't, stripping whitespace is no problem whatsoever. If you make a clear design that tells whether you're working with mixed content, you can avoid removing significant whitespace from elements.

Dealing with Whitespace in the Stylesheet

Whitespace doesn't have to originate in the source XML document. It also can originate from the stylesheet used to transform the source XML document. Whitespace-only nodes in XSLT are ignored and do not appear in the output. However, if you create mixed content by adding characters to the stylesheet, these characters are written to the output along with the surrounding whitespace. The result is most likely not the output you expected. Compare the code in Listings 7.19 and 7.20.

LISTING 7.19 Partial Stylesheet with Whitespace Only

```
<xsl:value-of select="@manufacturer" />
  <xsl:value-of select="@model" />
```

LISTING 7.20 Partial Stylesheet Containing Text

```
<xsl:value-of select="@manufacturer" />
  -<xsl:value-of select="@model" />
```

ANALYSIS The difference between Listings 7.19 and 7.20 is minor. In fact, only one character is different on the second line. Yet the output is rather different. The result from Listing 7.19 will not contain any spaces between the two values written, as shown here:

```
FordFocus
```

Listing 7.20, however, will produce output that contains all the whitespace in the stylesheet, as shown here:

```
Ford
  -Focus
```

As you can imagine, these results can be quite confusing when you're creating output. One moment all values written to the output have no whitespace in between, and the next moment new lines might appear in places where you don't want them. The best way to deal with this problem is to use the `xsl:text` element, which was specifically designed to deal with this situation.

The value of the `xsl:text` element might be only text with or without whitespace. It also might be only whitespace, but the basic idea behind the `xsl:text` element is that it cannot contain any element. The value of the `xsl:text` element is written to the output exactly as it appears in the stylesheet, even if it contains only whitespace. Listing 7.21 shows this use.

LISTING 7.21 Partial Stylesheet with Whitespace Only

```
<xsl:value-of select="@manufacturer" /><xsl:text> </xsl:text>
  <xsl:value-of select="@model" />
```

ANALYSIS The code in Listing 7.21 is somewhat different from the code in Listings 7.19 and 7.20. It is no longer mixed content, but rather has a separate element that contains a single space. The whitespace after the `xsl:text` element on the first line is now treated as a whitespace-only node. The whitespace value of the `xsl:text` element, however, is treated as if it were regular text. The whitespace-only node will not appear in the output, but the value of the `xsl:text` element will, so the output will look like this:

```
Ford Focus
```

The `xsl:text` element can have one attribute, `disable-output-escaping`, which can have the value `yes` or `no`. In the latter case, any text inside the element is sent to the output without it being output escaped. The same attribute can also be used to output a value with `xsl:value-of`.

Disabling output escaping is useful if you're creating output that has to conform to XML or HTML rules for the most part but which also contains sections that don't. An example is using XML and XSLT to generate Active Server Pages (ASP), Java Server Pages (JSP), or Personal Home Page (PHP) files, used in dynamic Web sites. These files contain HTML interspersed with script that is supposed to be executed on a Web server. In ASP and JSP, these code blocks start with `<%` and end with `%>`. To prevent these delimiters from coming out as `&lt;%` and `%&gt;`, you need to disable output escaping when writing these characters to the output.

As discussed earlier, the same problem also occurs with HTML files that must contain JavaScript code to be executed in the browser. Listing 7.22 shows part of an XML document that contains some JavaScript.

LISTING 7.22 Partial XML Document with JavaScript Code

```
1: <code language="javascript"><![CDATA[
2:   function maxval(A, B) {
3:     if(A < B) {
4:       return B;
5:     } else {
6:       return A;
7:     }
8:   }
9: ]]></code>
```

ANALYSIS Listing 7.22 contains a piece of script code that compares two numbers and returns the largest number. Line 3 contains a < character, which is illegal in regular XML. To keep the script code as is, the code is contained in a CDATA section, as you can see from the CDATA section delimiters on lines 1 and 9. Listing 7.23 is part of a stylesheet that inserts this script in an HTML document.

LISTING 7.23 Stylesheet Disabling Output Escaping

```
1: <xsl:template match="code">
2:   <script language="{@language}">
3:     <xsl:text disable-output-escaping="yes"><!--&#xA;</xsl:text>
4:       <xsl:value-of select="." disable-output-escaping="yes" />
5:     <xsl:text disable-output-escaping="yes">&#xA;--></xsl:text>
6:   </script>
7: </xsl:template>
```

ANALYSIS Listing 7.23 shows a template matching the `code` element in Listing 7.22. It inserts a script element on line 2 as is common in HTML. The language of the script block is retrieved from the `language` attribute of the `code` element. Lines 3 and 5 make sure that the code inside the `script` element is enclosed in a comment so that older browsers will not have a problem with the script code. Line 4 inserts the entire script and tells the processor to disable output escaping, so the special characters in the script are output as is. You can see that this is done correctly in the result in Listing 7.24.

Note

Lines 3 and 5 in Listing 7.23 could have used `xsl:comment` instead of `xsl:text` to insert the comment tags. For this sample, `xsl:text` is more appropriate.

OUTPUT **LISTING 7.24** Result from Applying Listing 7.23 to Listing 7.22

```
<script language="javascript">
<!--
  function maxval(A, B) {
    if(A < B) {
      return B;
    } else {
      return A;
    }
  }
-->
</script>
```

Indenting

Most of the sample listings in this and previous lessons are nicely indented so that you can easily see where an element starts and where it ends. These listings were all created by hand (except result listings, of course). When you're generating XML or HTML, you might want to use nicely indented code as well. The `xsl:output` element has an attribute to control indenting.

Note For the text output method, the indent attribute is ignored. It doesn't make much sense anyway because the output is indented based on XML elements, which text output doesn't contain.

This attribute, `indent`, can have the value `yes` or `no`. The value `no` is the default, in which case the processor just outputs the code as it encounters elements, text, and whitespace, often creating one large continuous string without any line feeds. If you use `indent="yes"`, however, the output looks quite nice. Listing 7.25 shows a sample of actual output using `indent="yes"` based on Listing 7.3.

OUTPUT **LISTING 7.25** Result Using `indent="yes"`

```
<?xml version="1.0" encoding="utf-8"?>
<form method="POST" action="post.asp">
   <INPUT type="radio" name="car" value="Focus"/>FordFocus<br/>
   <INPUT type="radio" name="car" value="Golf"/>VolkswagenGolf<br/>
   <INPUT type="radio" name="car" value="Camry"/>ToyotaCamry<br/>
   <INPUT type="radio" name="car" value="Civic"/>HondaCivic<br/>
   <INPUT type="radio" name="car" value="Prizm"/>ChevroletPrizm<br/>
</form>
```

ANALYSIS Listing 7.25 shows the result from applying Listing 7.3 to Listing 7.2, but with line 5 changed to

```
<xsl:output method="xml" version="1.0" indent="yes" />
```

The result is that the code is nicely indented. Other than that, the result has not really changed. In this case, it looks nice, but that's because no whitespace is involved. If the source document contains whitespace, how the output will be indented is not defined. The processor, in fact, isn't even obliged to indent the code at all, even if you have specified that it should. The only reason for the indents is that they generate a result that is easier for human eyes to read. Because the indents serve no other purpose, what the actual output should look like is up to the parser. What the actual output looks like will not affect how the output is dealt with down the road.

Summary

Today you learned that you can use XSLT to create different types of output. Apart from creating XML, you can create HTML or plain text documents. You can use the latter for comma-delimited data files, some character-encoded proprietary file formats, and many other things. By using an extra tool, you also can create PDF-, RTF-, and PCL-formatted documents using XSLFO. In the future, you might see processors that have such functionality embedded.

Apart from creating different output types, you can also determine the character encoding used to create documents. Parsers and processors support at least UTF-8 or UTF-16 encoding, and others are also possible (but not mandatory).

The last piece of control you need over the output is how and where whitespace is inserted. Whitespace can come from both the XML source document and the stylesheet, and there are several elements and functions to deal with whitespace. Handling whitespace can be tricky business because whitespace-only nodes can easily be removed when that is not what you want.

Q&A

Q Can I create binary file formats from XSLT?

A Unless there is a one-to-one mapping between characters and bytes, creating binary file formats from XSLT is not possible. The only way you can do so is to use tools or extensions such as FOP and jfor.

Q Can I insert data in a database with XSLT?

A You cannot insert data into a database with XSLT by itself. You can, however, create comma-delimited files or something similar and import those files into a database. Some databases offer functions that can import an XML file in a similar fashion.

Q The `xsl:strip-space` elements in my stylesheet are ignored at some points, but those elements don't use `xml:space="preserve"`. What's going on?

A Most likely an ancestor element of the element you are processing does have `xml:space="preserve"`. This attribute is used on all descendent elements of that element.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: `xsl:output` can be a child element of an `xsl:template` element.
2. True or False: `xsl:strip-space` and `xsl:preserve-space` are top-level elements.
3. What will happen if you are creating an XML document and the characters that you send to the output are not supported by the encoding type used? What would happen if you were creating a text document?
4. When using `xsl:strip-space` and `xsl:preserve-space`, which `elements` attribute would have a higher priority, `foo:*` or `car`?
5. When would you use `normalize-space` instead of `xsl:strip-space`?

Exercise

1. Create a stylesheet for Listing 7.2 with templates, `xsl:strip-space`, and `xsl:text` so that it has the following output:

```
List of cars
-Ford Focus (2000)
-Volkswagen Golf (1999)
-Toyota Camry (1999)
-Honda Civic (2000)
-Chevrolet Prizm (2000)
```

WEEK 1

In Review

In this first week, you have covered a lot of ground, from learning what XSLT is to developing the foundation needed to create complex stylesheets. That foundation is actually so complete that you can already create stylesheets with many different features. This capability is one of the things that makes XSLT so interesting because you can achieve a great deal with only a few elements and functions. In the next two weeks, you will learn about even more elements and functions, but much of that information is also about refining the knowledge gained in this first week.

Overview of Bonus Project 1

The samples and exercises of the lessons nearly all focus on one aspect of XSLT: an element, an attribute, or a function. Looking at XSLT this way is very important because you are not distracted by other functionality. The downside to this approach is that you don't really get an idea of how all the functionality interacts, as is the case in a real-world application. Each week concludes with a bonus project that shows you how to create a real-world application from beginning to end. These projects will focus on what you have learned in the past week.

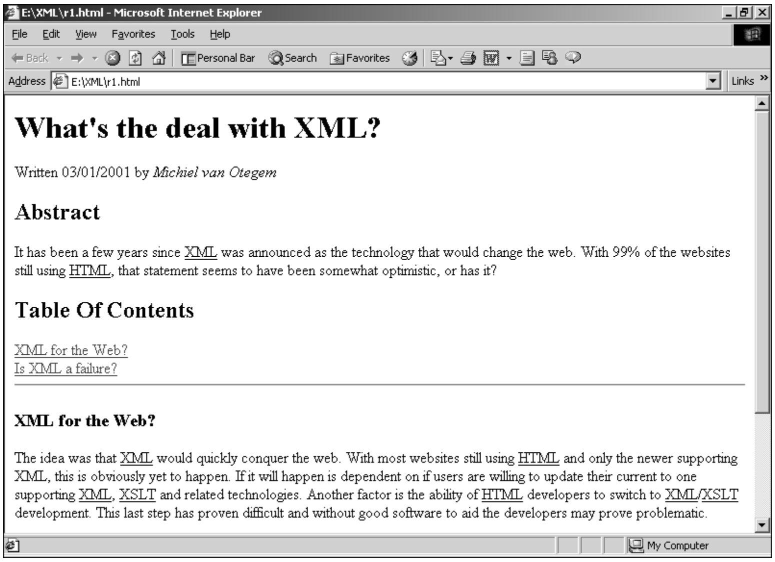
Creating an Article with a Table of Contents

On Day 1, "Getting Started with XSLT," I told you that one of the advantages of XML is that it can work with relatively unstructured data such as articles in a paper or magazine or on a Web site. When you have an article in XML, you can easily convert it to other output types, satisfying multiple needs. Bonus Project 1 is about creating a stylesheet for one such output—in this case, HTML. The aim of this project is to use as many of the elements and functions you have learned about in the past week as possible. Therefore, the XSLT document that results from this project will contain different solutions for similar problems. If this were a real-world project, you would probably try to use the same solution for the same types of problems because that would make the XSLT document easier to read and understand. This project, however, is a learning tool with the purpose of showing you the differences and similarities of different approaches. Let's get started!

Project Overview

In this project, two files, the source XML and the stylesheet used to process the source XML. You can create additional XML sources by using the same XML structure used for the source XML created in this project. The stylesheet is used to transform the XML source to HTML.

The resulting HTML should provide the article in a readable fashion to the reader. This means that you will have to create some kind of layout with headers, text, and possibly effects to mark special words. The article is also divided in sections with headers. At the start of the article, a table of contents should show all the article headers. To make the article easy to navigate, you will make these headers into hyperlinks linking to the anchor inserted at the point where the section starts. Figure BP1.1 gives you an idea of what this article will look like.


E:\XML\r1.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address E:\XML\r1.html
Links
My Computer

What's the deal with XML?

Written 03/01/2001 by *Michiel van Otegem*

Abstract

It has been a few years since XML was announced as the technology that would change the web. With 99% of the websites still using HTML, that statement seems to have been somewhat optimistic, or has it?

Table Of Contents

XML for the Web?
Is XML a failure?

XML for the Web?

The idea was that XML would quickly conquer the web. With most websites still using HTML and only the newer supporting XML, this is obviously yet to happen. If it will happen is dependent on if users are willing to update their current to one supporting XML, XSLT and related technologies. Another factor is the ability of HTML developers to switch to XML/XSLT development. This last step has proven difficult and without good software to aid the developers may prove problematic.

FIGURE BP1.1
Article marked up in HTML, as shown in a browser.

In Figure BP1.1, the article header is followed by information on when and by whom the article was written. Then it has some brief introductory text, an abstract, and a table of contents with headers as links. The article text starts below the horizontal line, with each section starting with a header. (In Figure BP1.1, you can see only the start of the first section.)

In this project, you will re-create the exact HTML resulting in the article in Figure BP1.1.

Creating the Article XML

Before you can create a stylesheet, you first need to have an XML structure to create the article or articles in. When you're developing this structure, it is important that the structure formed by the elements is logical. The relationship between the elements and what the elements represent should be clear. It is therefore also important that the names you choose for elements and attributes are representative of the information they contain. So, if you want to create an XML document containing an article, you would be wise to name the root element `article` and store the name or names of the person or persons who wrote it in an element called `author`. What is not a good idea is to name these elements `a1` and `a2`, or `ar` and `au`, because these element names may make sense to you but not to anybody else. You can use short names only when there is some kind of convention, such as using the name `para` for paragraph elements. Listing BP1.1 shows a small article with elements (and attributes) that you would typically find in an article.

LISTING BP1.1 Article Tagged in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<article id="xml0001" date="03/01/2001">
  <title>What&apos;s the deal with XML?</title>
  <author firstname="Michiel" lastname="van Otegem" />
  <intro>It has been a few years since <term>XML</term> was announced as
         the technology that would change the web. With 99% of the
         websites still using <term>HTML</term>, that statement seems to
         have been somewhat optimistic, or has it?
  </intro>
  <body>
    <para header="XML for the Web?">
      The idea was that <term>XML</term> would quickly conquer the web.
      With most websites still using <term>HTML</term> and only the newer
      <device>browsers</device> supporting XML, this is obviously yet to
      happen. If it will happen is dependent on if users are willing to
      update their current <device>browsers</device> to one supporting
      <term>XML</term>, <term>XSLT</term> and related technologies.
      Another factor is the ability of <term>HTML</term> developers to
      switch to <term>XML</term>/<term>XSLT</term> development. This last
      step has proven difficult and without good software to aid the
      developers may prove problematic.
```

LISTING BP1.1 Continued

```
22:     </para>
23:     <para header="Is XML a failure?">
24:       <term>XML</term> is most certainly not a failure. The fact that it
25:       hasn&apos;t caught on the front-end doesn&apos;t mean that it
26:       hasn&apos;t caught on at all. <term>XML</term> is very popular as a
27:       means of communicating data between systems and applications.
28:     </para>
29:     <para>
30:       Because <term>XML</term> is a standard way of communicating data
31:       and is capable of representing many data models, it is a natural
32:       choice when communication is needed between systems. Because
33:       <term>XML</term>is a string format any operating system can read
34:       it.
35:     </para>
36:   </body>
37: </article>
```

Note You can download the sample listings in this project from the publisher's Web site.

ANALYSIS The article in Listing BP1.1 is represented inside a root element called `article`. As you can see on line 2, this element has two attributes: the `id` attribute containing a unique identifier for this article and the `date` attribute containing the date when it was written. The `id` attribute is somewhat superfluous when you're storing articles as separate documents but may prove handy when you want to merge files into a larger document. The `article` element has several child elements with different functions, or rather with different types of data. The `title` and `author` elements on lines 3 and 4 are fairly obvious. The `title` element could just as well have been an attribute of the `article` element, but whether that is a good idea is arbitrary. There could, however, be more than one author, in which case you would need multiple elements, so having `author` as an attribute is not possible. The `firstname` and `lastname` attributes of the `author` element speak for themselves.

The `intro` element on lines 5 through 9 contains introductory text or an abstract about the article. Note that it can contain `term` elements to denote terms used in the text. These `term` elements are also used in the text that is stored in `para` elements, of which there are several on lines 11 through 36. They are child elements of the `body` element, which is included to denote the article's body text. A `para` element can have a `header` attribute, as shown on lines 11 and 23, but it is not mandatory. Such a `header` attribute contains the header for the `para` element it comes with and any following `para` elements that don't have a `header` attribute. Hence, the `header` attribute on line 11 serves as a header for the `para` element on line 11, and the `header` attribute on line 23 serves as a header for the `para` elements on both lines 23 and 31.

Note

The elements used in Listing BP1.1 serve as examples. You could easily extend the sample with more elements and attributes to suit your needs.

Creating the XSLT Document

Creating a stylesheet is best done step by step, slowly building the desired output. This process, of course, always starts with an empty stylesheet, preferably with an XML declaration. The next step is to create a template that matches any element. This template should be empty so that it creates no output. The purpose of this template is to override any built-in rules so that you don't get any unexpected output from elements you didn't match. You also need to define the output method and, if needed, any `xsl:strip-space` and `xsl:preserve-space` elements. Because you're creating HTML, whitespace handling is, strictly speaking, not necessary, but in this project the `article` and `body` elements will be stripped of space. The result of the efforts so far is shown in Listing BP1.2.

LISTING BP1.2 Stylesheet with Basic Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="article body" />

  <xsl:template match="*" />
</xsl:stylesheet>
```

Note

In this project, each code listing expands on Listing BP1.2. The code listings you can obtain from the publisher's Web site contain a complete stylesheet that you can run, so you can see the differences between using and not using certain elements.

ANALYSIS The stylesheet in Listing BP1.2 will generate no output. The template matching any element is invoked for the `article` element in the source, and that's that. Processing stops and no output is created. Generating output is the task of other templates to be added. The purpose of this stylesheet is to make sure you are not faced with any unexpected and unwanted output.

Note

In Listing BP1.2, `match="*"` matches any element. Because you are dealing only with elements here, this works fine. In cases in which matching is invoked for attributes and text, `match="/ | * | @* | text()"` is an alternative that is often used. The latter expression matches the root element, any element, any attributes, and text nodes.

Creating the HTML Base

You're creating HTML, so the output should contain the basic elements needed in HTML. The best place to insert them is a template matching the source document's root. After all, these elements are themselves root elements, but for an HTML document. Listing BP1.3 shows the root template.

LISTING BP1.3 Root Template Creating HTML Elements

```
<xsl:template match="/">
  <html>
    <body>
      <xsl:apply-templates />
    </body>
  </html>
</xsl:template>
```

ANALYSIS When the stylesheet encounters the root node, the template in Listing BP1.3 is matched and the `html` and `body` elements inserted. The `xsl:apply-templates` element makes sure that processing continues. At this point, this element will match the wildcard template in Listing BP1.2, so processing will stop there. This means that, for now, only the elements inserted here are part of the output.

Showing the Article Information

Now it's time to really create some output from the source document. Because the result should start with the article title and information on who wrote it and when, this is what you should concentrate on first. Because this information originates in the `article` element, getting it first is not a problem because it is the first element to be matched after the document root. A template matching the `article` element will therefore be matched when `xsl:apply-templates` is executed in Listing BP1.3. Listing BP1.4 shows the partial stylesheet responsible for the next phase of the output.

LISTING BP1.4 Partial Stylesheet Creating the Article

```
 1: <xsl:template match="article">
 2:   <xsl:call-template name="info" />
 3:   <h2>Abstract</h2>
 4:   <p><xsl:apply-templates select="intro" /></p>
 5:   <h2>Table Of Contents</h2>
 6:   <xsl:apply-templates mode="toc" />
 7:   <hr />
 8:   <xsl:apply-templates mode="body" />
 9: </xsl:template>
10:
11: <xsl:template name="info">
12:   <h1><xsl:value-of select="title" /></h1>
13:   <p>
14:     Written <xsl:value-of select="@date" />
15:     by <i>
16:       <xsl:value-of select="author/@firstname" />
17:       <xsl:text> </xsl:text>
18:       <xsl:value-of select="author/@lastname" />
19:     </i>
20:   </p>
21: </xsl:template>
```

ANALYSIS Listing BP1.4 may look a little complex, but it is not hard to understand. First, the template matching the `article` element consists of some output that is created inline, but also using both named and matched templates. On line 2, a named template is called to insert the article information. When you use a named template, the code is divided in more understandable pieces. The called template that starts on line 11 inserts the title of the article on line 12 and then proceeds to add the date and author. Everything is surrounded by HTML tags for layout. Note that on lines 16 and 18 the value that is selected doesn't come from the context node or a direct child node, but from attributes of a child element. Also, note that on line 17 an `xsl:text` element inserts whitespace. If that whitespace weren't inserted, the author's first name and last name would be inserted one after the other without any spaces in between.

Note

In this stylesheet, the author's name is inserted by referencing the `author` element. This means that if there is more than one author, only the first in the source document is inserted. If you want to have all of them, you need to use `xsl:for-each` or additional templates, and add some kind of delimiter. Adding this functionality is a good exercise after you finish Bonus Project 1.

After the template dealing with the article information is called, the template matching the `article` element continues to insert HTML elements. Additionally, on line 4 processing continues specifically on the `intro` element. On lines 6 and 8, processing continues, but in two separate modes, one for the table of contents (TOC) and one for the actual article text. The two separate modes are used because the `para` elements need to be processed once for the TOC and once for the article text.

Note

Instead of applying templates in two different modes, you could also use `xsl:for-each` to select the `para` elements that have a `header` attribute and output them. Then, when the article text needs to be created, you can use `xsl:apply-templates` without any mode.

Because you're using templates with modes, you're inserting a new factor that can cause side effects, such as inserted text, because an element isn't matched and the default template rule takes over. To make sure that you don't have any side effects, you need to add empty templates for those modes, just like you did for the default mode. These templates are shown in Listing BP1.5.

LISTING BP1.5 Templates Preventing Side Effects

```
<xsl:template match="*" mode="toc" />
<xsl:template match="*" mode="body" />
```

Inserting the Abstract

In Listing BP1.4, you could see that the `intro` element was specifically selected when applying new templates. To have it inserted into the output, you need to add templates that actually add it. Listing BP1.6 shows the code that inserts this element.

LISTING BP1.6 Code Responsible for Inserting the Abstract

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

<xsl:template match="term">
  <u><xsl:value-of select="." /></u>
</xsl:template>

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

ANALYSIS The template on line 1 that matched the `intro` attribute does nothing more than re-invoke the processor for its child elements. Because the value of the `intro` element contains both text and elements, both the child elements and text nodes need to be matched. In the sample article, the only child elements are `term` elements, which are matched by the template on line 5. This template inserts the value of those elements and underlines them using HTML underline tags. The template on line 9 matches any text and just inserts it into the output.

Note

> Because the output is HTML, stripping whitespace is not necessary. If you stripped whitespace from the text nodes using the `normalize-space()` function, you would inadvertently remove significant whitespace.

Inserting the Table of Contents

The next step in the output is the table of contents (TOC). On line 6 of Listing BP1.4, you saw that a separate mode was invoked to create the TOC. This means that the templates you create to insert the TOC need to use this mode; otherwise, it will not appear in the output. Because there is already a template dealing with any element that is not matched by a specific template in this mode, you need to add templates only for the elements that need to be matched. They are any child nodes of the context element, which is the `article` element. However, for the TOC, you need to match only the `body` element and its `para` child elements, as shown in Listing BP1.7.

LISTING BP1.7 Template Creating the Table of Contents

```
1:  <xsl:template match="body" mode="toc">
2:    <xsl:apply-templates mode="toc" />
3:  </xsl:template>
4:
5:  <xsl:template match="para" mode="toc">
6:    <xsl:if test="@header">
7:      <a href="#{@header}">
8:        <xsl:value-of select="@header" />
9:      </a><br /><xsl:text>&#xA;</xsl:text>
10:   </xsl:if>
11: </xsl:template>
```

ANALYSIS The template on line 1 in Listing BP1.7 matches the `body` element. It invokes the processor again only to match its child elements. You may think that because of the built-in template rule, it is not necessary, as the child elements will be processed by the built-in rule if the `body` element is not matched. The template added in Listing BP1.5 also matches the `body` element, however, and would stop any further processing.

The template on line 5 matches the `para` elements. The code in that template is a little more elaborate because it needs to filter out any unwanted paragraphs. The only paragraphs for which something needs to be inserted into the TOC are those that have `header` attributes. Line 6 tests whether the current `para` element has a `header` attribute. If there is no attribute, the code inside the `xsl:if` element is not executed. If there is a `header` attribute, a hyperlink is inserted. The value of this hyperlink is, of course, the header itself; the URL to which the hyperlink points actually points to an HTML anchor in the same document. This is achieved with line 7, which creates an `a` element with an `href` attribute. The value of the attribute is part static and part dynamic. The dynamic part is enclosed in curly braces and inserts the value of the `header` attribute.

What the output looks like in a browser will not be influenced by what the actual source output will look like. To make that source is a little more readable to somebody requesting it, however, a linefeed is inserted on line 9.

Inserting the Article Body

Last but not least, the article body should be inserted into the output. This is, after all, what the entire output is about—no article body…nothing to read. The article body again uses a mode to separate it from other types of sections that need to be inserted into the output. The code is shown in Listing BP1.8.

LISTING BP1.8 Template Inserting the Article Body

```
1:  <xsl:template match="body" mode="body">
2:    <xsl:for-each select="para">
3:      <xsl:if test="@header">
4:        <a name="{@header}">
5:          <h3><xsl:value-of select="@header" /></h3>
6:        </a>
7:        <xsl:text>&#xA;</xsl:text>
8:      </xsl:if>
9:      <p><xsl:apply-templates /></p>
10:   </xsl:for-each>
11: </xsl:template>
```

ANALYSIS The `body` element contains only (`para`) child elements, so you would expect the template in Listing BP1.8 to invoke the processor to match its child elements. Instead, I chose to use an `xsl:for-each` element that selects and processes each `para` element. The code inside the `xsl:for-each` element looks a lot like the code use earlier for the TOC items. Instead of creating a hyperlink, however, line 4 now creates an HTML anchor. Additionally, it doesn't matter whether the `para` element contains a `header` attribute. When it's time to insert the contents of the `para` element, `xsl:apply-templates` is used on line 9 to continue processing. The contents of the `para` element are similar to that of the `intro` element discussed earlier, which is why on line 9 no mode is used to match templates. The result is that the templates in Listing BP1.6 are matched, reusing that code.

If you now apply the entire stylesheet as shown in Listing BP1.9 to Listing BP1.1, you get the output that results in Figure BP1.1. Try it. As I noted earlier, many of the tasks performed in the stylesheet can be achieved in several other ways. It is a good exercise to try to replace some of the code with different code doing the same thing.

LISTING BP1.9 Complete Stylesheet for Bonus Project 1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="article body" />

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="article">
    <xsl:call-template name="info" />
    <h2>Abstract</h2>
    <p><xsl:apply-templates select="intro" /></p>
    <h2>Table Of Contents</h2>
    <xsl:apply-templates mode="toc" />
    <hr />
    <xsl:apply-templates mode="body" />
  </xsl:template>

  <xsl:template name="info">
    <h1><xsl:value-of select="title" /></h1>
    <p>
      Written <xsl:value-of select="@date" />
      by <i>
        <xsl:value-of select="author/@firstname" />
        <xsl:text> </xsl:text>
        <xsl:value-of select="author/@lastname" />
      </i>
    </p>
  </xsl:template>

```

LISTING BP1.9 Continued

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

  <xsl:template match="term">
    <u><xsl:value-of select="." /></u>
  </xsl:template>

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

  <xsl:template match="body" mode="toc">
    <xsl:apply-templates mode="toc" />
  </xsl:template>

  <xsl:template match="para" mode="toc">
    <xsl:if test="@header">
      <a href="#{@header}">
        <xsl:value-of select="@header" />
      </a><br /><xsl:text>&#xA;</xsl:text>
    </xsl:if>
  </xsl:template>

  <xsl:template match="body" mode="body">
    <xsl:for-each select="para">
      <xsl:if test="@header">
        <a name="{@header}">
          <h3><xsl:value-of select="@header" /></h3>
        </a>
        <xsl:text>&#xA;</xsl:text>
      </xsl:if>
      <p><xsl:apply-templates /></p>
    </xsl:for-each>
  </xsl:template>

  <xsl:template match="*" />
  <xsl:template match="*" mode="toc" />
  <xsl:template match="*" mode="body" />
</xsl:stylesheet>
```

Listing BP1.10 shows the result from applying Listing BP1.9 to Listing BP1.1.

Note

For your understanding of each section of code added, you should execute the code and look at the intermediate results as well.

OUTPUT **LISTING BP1.10** Result When Listing BP1.9 Is Applied to Listing BP1.1

```
<html>
   <body>
      <h1>What's the deal with XML?</h1>
      <p>
         Written 03/01/2001
         by <i>Michiel van Otegem</i></p>
      <h2>Abstract</h2>
      <p>It has been a few years since <u>XML</u> was announced as the
         technology that would change the web. With 99% of the websites still
         using <u>HTML</u>, that statement seems to have been somewhat
         optimistic, or has it?

      </p>
      <h2>Table Of Contents</h2><a href="#XML for the Web?">XML for the
➥ Web?</a><br>
      <a href="#Is XML a failure?">Is XML a failure?</a><br>

      <hr><a name="XML for the Web?">
         <h3>XML for the Web?</h3></a>

      <p>
         The idea was that <u>XML</u> would quickly conquer the web. With
         most websites still using <u>HTML</u> and only the newer
         supporting XML, this is obviously yet to
         happen. If it will happen is dependent on if users are willing to
         update their current  to one supporting
         <u>XML</u>, <u>XSLT</u> and related technologies. Another
         factor is the ability of <u>HTML</u> developers to switch to
         <u>XML</u>/<u>XSLT</u> development. This last step has
         proven difficult and without good software to aid the developers may
         prove problematic.

      </p><a name="Is XML a failure?">
         <h3>Is XML a failure?</h3></a>

      <p>
         <u>XML</u> is most certainly not a failure. The fact that it
         hasn't caught on the front-end doesn't mean that it
         hasn't caught on at all. <u>XML</u> is very popular as a
         means of communicating data between systems and applications.

      </p>
      <p>
         Because <u>XML</u> is a standard way of communicating data
         and is capable of representing many data models, it is a natural
         choice when communication is needed between systems. Because
         <u>XML</u> is a string format any operating system can read
         it.
```

LISTING BP1.10 Continued

```
        </p>
    </body>
</html>
```

WEEK 2

At a Glance

Last week you learned the foundation of XSLT. With that foundation, you can already do quite a bit, but you are also bound to that foundation, which still lacks flexibility. This week you concentrate on learning the elements and functions that give you more flexibility and power.

On Day 8, "Working with Variables," you will learn how variables can aid you in making complex tasks easier. You also will learn that variables are great for storing information that you need throughout a stylesheet.

Day 9, "Working with Parameters," builds on Day 8 and teaches you about the big brother of variables: parameters. Parameters are all about flexibility because they allow you to alter the processing of a template based on a parameter value. This way, you don't have to create different templates for tasks that differ only slightly. In addition, you can make templates into *functions* that return a value.

Day 10, "Understanding Data Types," is all about the basic data that is stored in elements and attributes. Data types tell you how the data will be interpreted by the processor and what will happen. This lesson also deals with converting one data type into another.

Day 11, "Working with Strings," is closely related to Day 10. It is all about the key data type in XSLT: the string. Everything in XSLT is a string to start with, so your ability to fully understand strings and manipulate them is very important.

On Day 12, "Sorting and Numbering," you will learn how to sort data in an XML source. Sorting can be both static, on a predetermined value, or dynamic, so you can determine the sort order at runtime. You also will learn how to number elements. You can use simple numbers, but also letters, Roman numerals, and composite numbers.

Day 13, “Working with Multifile Stylesheets,” is all about making your job as a programmer a little easier. It teaches you how to build stylesheets from reusable pieces and how to use another stylesheet and alter it to meet your needs.

Finishing Week 2 is Day 14, “Working with Multiple XML Sources.” Just like you can break down a stylesheet into separate pieces, you can do the same with some XML structures. If data is also to be used separately, it may benefit from being in a separate file. But how do you apply a stylesheet to data in multiple files? That question is answered on Day 14.

After you finish Week 2, you will have expanded your knowledge of XSLT so that you can really start to build useful applications. You also will have more insight into why some XSLT features work the way they do.

WEEK 2

DAY 8

Working with Variables

Yesterday you learned to manipulate the actual output that is created with elements and functions covered in the days before it. You learned how to determine the output type and how to control whitespace. In a sense, yesterday's lesson had nothing to do with processing, but rather how the processed output actually came out of the processor.

Today's lesson again covers the processing side of XSLT, specifically how to deal with complex situations in which expressions can become very complex, so complex that they are hard to create and understand. Another topic that will be revisited is the way the current context affects expressions and ways to get around that problem.

Today you will learn the following:

- What variables are
- How to create and use variables
- How variables can help you to make stylesheets less complex
- How variables solve problems with the current context

Understanding Variables

In all the stylesheets you've created so far, you have been operating on the source XML document as a whole, based on the current context. That situation is not ideal if you want to create complex output, in which some of the output might depend on data that is out of context. By that, I mean operating on the output without a complex expression is not easy, and in some cases, it's hardly possible. Variables are a mechanism in XSLT designed to help in these situations.

What Are Variables?

If you have experience with programming, you're probably thinking, "Hey, I know what variables are.…" If that's the case, you're in for a surprise. Variables in most programming languages and variables in XSLT are not quite the same. They have similarities but some radical differences as well because variables in XSLT are more closely related to mathematical variables than variables in common programming languages. In a mathematical context, a variable is nothing more than a name given to a certain value. If you, for instance, have an equation such as

```
3x - 9 = 0
```

`x` is a variable. The term *variable* in such an equation is a bit misleading because this equation has only one solution: `x` is 3. So, the variable `x` just denotes the value 3 and cannot change unless the whole equation is changed.

Variables in XSLT are similar to mathematical variables in that they can be declared and given a value. After that value is given, the variable can no longer change; its value is fixed. In contrast, a variable in common programming languages can be changed after it has been declared and given a value, as done in the following code:

```
var x = 3;
var y = 4;
document.write(x);
x = y * 4;
document.write(x);
```

This code declares two variables and gives them a value. Then it writes the value of variable `x to the output`. Next, the variable `x` is given a new value, which is subsequently written to the output again. So, you see that a variable in this case does not have a fixed value. In fact, many programming constructs rely on this principle. A good example is a code snippet shown on Day 6, "Conditional and Iterative Processing," which is as follows:

```
For i = 1 To 10
    'execute some code
Next
```

This code contains a variable `i`, which increases in value each time the code iterates. In this case, the variable increases 10 times, from 1 to (and including) 10.

Programming constructs like the ones shown in the preceding examples are not available in XSLT. Part of the reason is that XSLT is a declarative programming language: You tell the processor what you want, not how you want it to happen. When you tell the processor that you want each element in a node-set processed, it does so, no matter how many nodes are in the node-set. Also, you don't have to figure out how many nodes there are and then tell the processor to do something a certain number of times, giving it the data you want it to process in each iteration. So, when XSLT was designed, there was good reason to leave out these constructs.

What Is the Benefit of Variables?

Because variables in XSLT are different from variables in other languages, you might question their usefulness. However their use is different does not mean they are insignificant. In the following paragraphs, I'll describe most of the advantages of variables. This discussion is still a theoretical overview, but later in this lesson, you will see these points in action.

Variables can make your code much easier to read. Instead of repeating a complex expression to select a value, you can use that expression once to create a variable with a meaningful name. Wherever you need to use the value from the expression, you now can use the variable instead. Using variables this way, of course, makes your code much easier to read and understand because you quickly know from the variable name what you are dealing with. If, instead, you need to use the complex expression, figuring out what is actually happening might prove difficult. Another benefit of this approach is that you can break a complex expression into pieces and build it in steps using several variables. With each step, you know exactly what you are dealing with, because the expression in each stage is much simpler and therefore much less error prone.

If you reuse a variable many times in a stylesheet, it also is likely to increase the performance of that stylesheet, particularly when the expression you use is complex and the resulting value is a node-set or tree fragment. Most processors store a reference or references to the node or nodes that are the result of an expression that is stored in a variable. This means that the expression does not have to be evaluated again when the variable is used, probably saving execution time.

Performance Timing

Most processors enable you to measure the performance of your stylesheet with an XML source. This capability is offered as part of the API, but also as a command-line option. It is a good idea to familiarize yourself with the options of your processor of choice and to test different scenarios when you create stylesheets.

With MSXSL, you can get timing information by using the `-t` option. You can run it from the command line as follows:

```
MSXSL source.xml stylesheet.xsl -t
```

Saxon uses the same command-line option as MSXSL, so you can run Instant Saxon as follows:

```
SAXON -t source,xml stylesheet.xml
```

Or you can run full Saxon as follows:

```
java com.icl.saxon.StyleSheet -t source.xml stylesheet.xsl
```

With Xalan, you get timing information when you use the `-DIAG` option, so you can run it as follows:

```
java org.apache.xalan.xslt.Process -IN source.xml -XSL
➥stylesheet.xsl -DIAG
```

You also can use variables to define values in a central location, so you can change the values easily. For example, you might use some value for formatting a particular type of element. Although, in many cases, you also could use attribute-sets to do so, in some cases, this approach is easier or more beneficial.

Finally, you can use variables to hold values of nodes that are relevant to the current operation but are not accessible from the context node. This is particularly the case when you have nested operations in which the context node changes along with the nesting. You might need to operate on nodes that are not in context quite often, especially if you have to compare values of different nodes with the value of the context node. Without the ability to store the value of a node that is inaccessible from the current context, such a comparison might be hard to achieve.

Creating and Using Variables

With some of the theory of variables under your belt, it's time to look at the actual thing. As this section progresses, the different uses discussed in the preceding section will become more apparent.

Using Simple Variables

As I said, you can create variables that contain a node-set or tree fragment. Because these types of variables are harder to work with than variables containing only a single value, the focus will first be on the latter. I will refer to variables containing a single value, such as a string or Boolean value, as *a simple variable*.

A simple variable is most likely used to insert values at several places in a document, with a central location to change that value. You can create a simple value in two different ways. The easiest method is to use the following:

```
<xsl:variable name="somename">somevalue</xsl:variable>
```

You also can create the same variable by using the following:

```
<xsl:variable name="somename" select="'somevalue'" />
```

The apostrophe characters in the second method are needed to tell the processor that the value is a string. If you leave out the apostrophes, the value of the variable will likely be empty because you are telling the processor that it needs to use the value from the `somevalue` element within the current context. In all likelihood, that element does not exist. I strongly recommend using the former method because mistakes are hardly possible with it. Throughout this book, I will use the former method wherever it is applicable.

After you create a variable, you can access it through its name, preceded by a `$` character. So, to output the value of a variable, you use the following code:

```
<xsl:value-of select="$somename" />
```

The best way to see how a variable fits in a stylesheet is to look at an example. Consider Listing 8.1, which is a listing you should almost know by heart by now.

LISTING 8.1 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

Suppose you want to create an HTML table from the XML source in Listing 8.1, and you want the row colors to alternate. You could, of course, hard-code color codes into the template dealing with the creation of the table rows. If you have an elaborate site layout, however, those colors are likely to be used in other sections of the layout as well—for instance, in a frame around the table or page. If you want to change those colors, you have to go into each template that uses the colors and physically change them. If the colors are stored in variables, you have to change the variables in only one central location. Listing 8.2 shows a stylesheet that uses variables to store the colors.

LISTING 8.2 Stylesheet Using Variables for Color Management

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" />

  <xsl:variable name="bgcolor">#cccccc</xsl:variable>
  <xsl:variable name="altbgcolor">#ffffff</xsl:variable>

  <xsl:template match="/">
    <html>
    <body bgcolor="{$bgcolor}">
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="cars">
    <table bgcolor="{$altbgcolor}" width="75%">
      <xsl:for-each select="car">
        <tr>
          <xsl:attribute name="bgcolor">
            <xsl:choose>
              <xsl:when test="position() mod 2 = 0">
                <xsl:value-of select="$altbgcolor" />
              </xsl:when>
              <xsl:when test="position() mod 2 = 1">
                <xsl:value-of select="$bgcolor" />
              </xsl:when>
            </xsl:choose>
          </xsl:attribute>
          <xsl:call-template name="car" />
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

  <xsl:template name="car">
    <td><xsl:value-of select="@model" /></td>
    <td><xsl:value-of select="@manufacturer" /></td>
    <td><xsl:value-of select="@year" /></td>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS **NEW TERM** In Listing 8.2, two variables are created on lines 7 and 8: `bgcolor` and `altbgcolor`. The variables contain color values for both background color and alternate background color used when the `table` element is inserted on line 19, and again on lines 25 and 28 with the `tr` element. Note that when these elements are created, the attribute values are given inside `curly` braces because they are dynamic values based on an expression. That expression is only a variable, but it is an expression, nonetheless. The variables are created as top-level elements. This means that these variables are global. A *global variable* is available in the entire document.

On line 12, the variable `bgcolor` gives a value to the `bgcolor` attribute of the `body` element. The variable is inserted between curly braces to make sure it is processed as a variable, not as plain text. After the variable is processed, the output will contain the value of the variable, not `$somename`. On line 19, the variable `altbgcolor` is used in the same way to give a value to the `bgcolor` attribute of the `table` element.

Lines 22–31 are also significant. In that part of the code, a `bgcolor` attribute for the `tr` element on line 21 is created dynamically. For each `car` element in the source XML, a `tr` element is created. The value of the `bgcolor` attribute depends on whether the position of the context node within the node-set, selected by the `xsl:for-each` element on line 20, is even or odd. This is tested using the `position() mod 2 = 0` expression on line 24, which returns `true` if `position()` returns an even number. Its counterpart on line 27 returns `true` if `position()` is odd. When the value of `position()` is even, the value of the `bgcolor` attribute is the value of the `altbgcolor` variable, which is inserted on line 25. Otherwise, line 28 inserts the value of the `bgcolor` variable.

Line 32 calls a template to insert the cells in the table row. This template is called so that the template matching the `cars` element deals as much with inserting the differently colored table rows as possible. The result from applying Listing 8.2 to Listing 8.1 is shown in Listing 8.3.

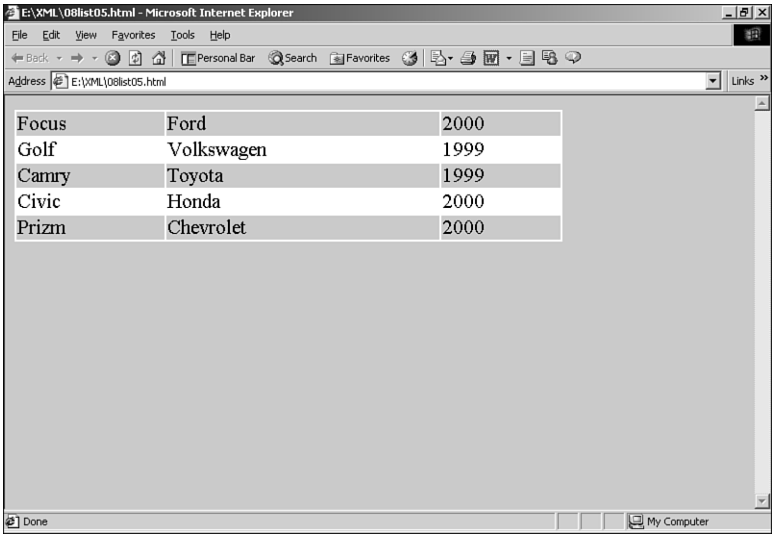
OUTPUT **LISTING 8.3** Result from Applying Listing 8.2 to Listing 8.1

```
<html>
   <body bgcolor="#cccccc">
      <table bgcolor="#ffffff" width="75%">
         <tr bgcolor="#cccccc">
            <td>Focus</td>
            <td>Ford</td>
            <td>2000</td>
         </tr>
         <tr bgcolor="#ffffff">
            <td>Golf</td>
            <td>Volkswagen</td>
            <td>1999</td>
         </tr>
```

Listing 8.3 Continued

```
14:          <tr bgcolor="#cccccc">
15:             <td>Camry</td>
16:             <td>Toyota</td>
17:             <td>1999</td>
18:          </tr>
19:          <tr bgcolor="#ffffff">
20:             <td>Civic</td>
21:             <td>Honda</td>
22:             <td>2000</td>
23:          </tr>
24:          <tr bgcolor="#cccccc">
25:             <td>Prizm</td>
26:             <td>Chevrolet</td>
27:             <td>2000</td>
28:          </tr>
29:       </table>
30:    </body>
31: </html>
```

Analysis In Listing 8.3, the `bgcolor` attribute of the `body` element on line 2 holds the value of the `bgcolor` variable from Listing 8.2. Also, the `table` element on line 3 has a `bgcolor` attribute with the value of the `altbgcolor` variable in Listing 8.2. Finally, the table rows have alternating background colors which you can see even more clearly in Figure 8.1.


E:\XML\08list05.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address E:\XML\08list05.html
Links
Done
My Computer

Focus	Ford	2000
Golf	Volkswagen	1999
Camry	Toyota	1999
Civic	Honda	2000
Prizm	Chevrolet	2000

Figure 8.1

The result in Listing 8.3 when viewed in a browser.

Simple variables are handy tools to reduce work when you have to change certain values in your output. You can create elements and attributes with certain values, or even elements and attributes themselves, based on a simple variable value. This way, your stylesheets can be very flexible.

Using Complex Variables

As I said earlier, you are not limited to using simple variables. They also can contain a node-set or tree fragment. I will therefore refer to them as *complex variables* as opposed to single-valued simple variables. Complex variables can be defined within a stylesheet itself, just like the `bgcolor` variable in Listing 8.2, but they also can be created with a select expression selecting a node-set or tree fragment in the source XML. This section will concentrate on defining variables in a stylesheet. Later in this lesson, you will learn how to create variables using a select expression.

Complex variables are particularly useful when you want to group values together. You can easily create such variables just by creating an XML structure inside the `xsl:variable` element. You cannot use the `select` attribute of the `xsl:variable` element to create such a value, as is possible with simple values, because the `select` attribute is incapable of holding XML data. You can use the `select` attribute only to create simple variables or variables containing data from the XML source created from a select expression. Listing 8.4 shows how to create and use a complex variable.

LISTING 8.4 Stylesheet with a Complex Variable

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" />

  <xsl:variable name="bgcolor">
    <body>#cccccc</body>
    <table>#ffffff</table>
    <row>#cccccc</row>
    <altrow>#ffffff</altrow>
  </xsl:variable>

  <xsl:template match="/">
    <html>
    <body bgcolor="{$bgcolor/body}">
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

```

LISTING 8.4 Continued

```
22:   <xsl:template match="cars">
23:     <table bgcolor="{$bgcolor/table}" width="75%">
24:       <xsl:for-each select="car">
25:         <tr>
26:           <xsl:attribute name="bgcolor">
27:             <xsl:choose>
28:               <xsl:when test="position() mod 2 = 0">
29:                 <xsl:value-of select="$bgcolor/altrow" />
30:               </xsl:when>
31:               <xsl:when test="position() mod 2 = 1">
32:                 <xsl:value-of select="$bgcolor/row" />
33:               </xsl:when>
34:             </xsl:choose>
35:           </xsl:attribute>
36:           <xsl:call-template name="car" />
37:         </tr>
38:       </xsl:for-each>
39:     </table>
40:   </xsl:template>
41:
42:   <xsl:template name="car">
43:     <td><xsl:value-of select="@model" /></td>
44:     <td><xsl:value-of select="@manufacturer" /></td>
45:     <td><xsl:value-of select="@year" /></td>
46:   </xsl:template>
47: </xsl:stylesheet>
```

ANALYSIS Listing 8.4 isn't much different from Listing 8.2. However, instead of creating multiple variables, one variable called `bgcolor` is created on line 7. It contains several elements, with colors defined for the different HTML elements, similar to a Cascading Stylesheet. On line 16, the `bgcolor` attribute is now set to the value of the `body` element within the `bgcolor` variable. Similarly, on line 23, the background color of the table is set to the value of the `table` element in the variable. Lines 29 and 32 insert the alternating background colors for the table rows.

You can create multiple variables, both simple and complex. Using complex variables, you can create groups of values that are used for the same purpose or for the same element or elements. In some cases, you could use attribute-sets to achieve the same result. Whether variables or attribute-sets are the best solution depends on the situation.

Creating Variables from Expressions

In the preceding sections, you learned how to create both simple and complex variables by hard-coding their values in the stylesheet. This way, you can store information relevant to the transformation in a central location, which enables you to easily edit common values. In some situations, you need to create variables that are based on the source XML. Such values can again be simple values, but also node-sets or tree fragments, depending on the select expression used. You can create these variables only by using the `select` attribute of the `xsl:variable` element. The expression used in the `select` attribute is used to select data from the source XML.

Using Variables to Replace an Expression

Expressions addressing data in the XML source can become very complex and very long. If that is the case, storing an expression's result in a variable is often helpful, especially if you have to use that expression in more than one place. If you give the variable a descriptive name, everybody can understand what the expression selects, without having to read or understand the expression itself. As I said previously, the processor does not have to evaluate the expression again if the variable is used again, so there also might be a slight performance benefit.

On Day 3, "Selecting Data," you learned how to create a nice-looking menu from an XML source with the data for that menu. Listing 8.5 shows the source XML used on Day 3.

LISTING 8.5 Sample XML Source with Menu Data

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.75">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.75">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolate Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Suppose you want to create a menu that also shows *this week's menu*, which has to vary every week. You could create this menu by hand, but you also could automate the task somewhat. If you want to do that, the expressions very likely will become quite complex. Listing 8.6 shows that this is indeed the case and that you can use variables to make the stylesheet less complex.

LISTING 8.6 Stylesheet Using Selected Variable

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" />
6:
7:    <xsl:variable name="week">7</xsl:variable>
8:
9:    <xsl:variable name="weekappetizer"
10:        select="/menu/appetizers/dish[position() =
11:        ((($week - 1) mod count(/menu/appetizers/dish)) + 1)]" />
12:
13:   <xsl:variable name="weekentree"
14:        select="/menu/entrees/dish[position() =
15:        ((($week - 1) mod count(/menu/entrees/dish)) + 1)]" />
16:
17:   <xsl:variable name="weekdessert"
18:        select="/menu/desserts/dish[position() =
19:        ((($week - 1) mod count(/menu/desserts/dish)) + 1)]" />
20:
21:  <xsl:template match="/">
22:     <xsl:text>This week's menu:&#xA;</xsl:text>
23:     <xsl:text>- </xsl:text><xsl:value-of select="$weekappetizer" />
24:     <xsl:text> $</xsl:text><xsl:value-of select="$weekappetizer/@price" />
25:     <xsl:text>&#xA;- </xsl:text><xsl:value-of select="$weekentree" />
26:     <xsl:text> $</xsl:text><xsl:value-of select="$weekentree/@price" />
27:     <xsl:text>&#xA;- </xsl:text><xsl:value-of select="$weekdessert" />
28:     <xsl:text> $</xsl:text><xsl:value-of select="$weekdessert/@price" />
29:   </xsl:template>
30: </xsl:stylesheet>
```

ANALYSIS Listing 8.6 defines four variables, one with a hard-coded value and the others defined from an expression depending on both the source XML and the hard-coded variable. On line 7, a variable named `week` is defined and given a value. This value is the number of the current week within the year. As you can see, the current week is defined as week 7. On line 9, a variable named `weekappetizer` is created using a complex expression that calculates which appetizer is this week's appetizer, based on the number in the `week` variable. On lines 13 and 17, the same thing is done to get an entree and a dessert. The three variables that result from these expressions are used in the template on line 21. Each variable is used twice, once to get the name of the dish and once to get the price. If you didn't have variables, you would have to use the expressions used to create the variables in both instances, which would make the template more complex. Also, with the variables, you can easily understand what is happening because the variables have understandable names. Without variables, it is much harder to grasp what's actually going on.

Let's look at the expressions for a moment to see what's actually happening there. I'll concentrate on the expression creating the `weekappetizer` variable, but the idea is the same for all of them. The expression selects all `dish` elements that are child elements of the `appetizers` element. A predicate is used to select the `dish` element that is the appetizer for this week. The number is calculated by taking the current week and dividing that by the number of appetizers. The resulting number is the appetizer for this week. The number of appetizers is calculated with the `count()` function, which returns the number of nodes in the given node-set. The `mod` operator determines the remaining number after the number to the left has been divided by the number to the right. Both of these numbers are the result of an expression, so it looks more complex than it is. To make sure that week 1 will always yield a menu that consists of the first appetizer, the first entree, and the first dessert in the source document, the expression is adjusted by adding or subtracting 1. The result of the expression is that on week 1 the first appetizer will be selected; on week 2, the second appetizer; and so on. On week 5, when all appetizers have been selected once, the selection starts at the beginning of the list again. The same expression is used for the entrees and desserts, and because there are four appetizers, five entrees, and three desserts, the menu will be unique for many weeks to come.

If the expressions are still too complex for your taste, you can divide them into pieces as well, as shown in Listing 8.7.

LISTING 8.7 Replacement Code for Lines 9–11 in Listing 8.6

```
<xsl:variable name="correctedweek" select="$week - 1" />
<xsl:variable name="appetizercount"
     select="count(/menu/appetizers/dish)) + 1" />
<xsl:variable name="appetizerpos" select="$correctedweek
➥mod $appetizercount" />
<xsl:variable name="weekappetizer"
     select="/menu/appetizers/dish[$appetizerpos]" />
```

The result of the preceding code is the same as lines 9–11 of Listing 8.6. Because the expressions are broken into smaller pieces, the complexity of the expression is as well. Each step of building the result is easy to understand, and the result is the same. That said, if you are able to understand and write the original expression, your code is cleaner because it isn't cluttered with variables. That, in itself, might make your code more readable. When and if you break up an expression is a matter of judgment.

Apart from Listing 8.6 being easier to read, it is very likely that using variables is faster because the expressions have to be evaluated only once, instead of once for the name of the dish and once for the price. Applying Listing 8.6 to Listing 8.5 yields the result in Listing 8.8.

OUTPUT **LISTING 8.8** Result from Applying Listing 8.6 to Listing 8.5

```
This week's menu:
- Smoked Salmon and Avocado Quesadilla $10.95
- Seafood Pasta $17.95
- Dame Blanche $6.95
```

ANALYSIS In Listing 8.8, week 7 yields the third appetizer, the second entree, and the first dessert from Listing 8.5. If you take the time to count through each week, you will find that this is exactly the output that should be created.

Using Variables for Out-of-Context Data

When you're working with templates and iterations, you likely will be nesting elements, digging deeper into the XML source. As long as you're digging into one part of the XML tree structure, there is no problem in addressing elements higher up in the nesting structure. However, if you have two related structures that do not have a parent-child or parent-descendant relationship, just selecting data from a location higher in the nesting structure does not yield the expected result. To make this point clearer, look at Listing 8.9.

LISTING 8.9 Sample XML Document with Separate Data Sets

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <models>
    <model name="Golf" manufacturer="VW" year="1999" />
    <model name="Camry" manufacturer="TY" year="1999" />
    <model name="Focus" manufacturer="FO" year="2000" />
    <model name="Civic" manufacturer="HO" year="2000" />
    <model name="Prizm" manufacturer="CV" year="2000" />
    <model name="Celica" manufacturer="TY" year="2000" />
    <model name="Mustang" manufacturer="FO" year="2001" />
    <model name="Passat" manufacturer="VW" year="2001" />
    <model name="Accord" manufacturer="HO" year="2002" />
    <model name="Corvette" manufacturer="CV" year="2002" />
  </models>
  <manufacturers>
    <manufacturer id="VW" name="Volkswagen" country="Germany" />
    <manufacturer id="TY" name="Toyota" country="Japan" />
    <manufacturer id="FO" name="Ford" country="USA" />
    <manufacturer id="CV" name="Chevrolet" country="USA" />
```

```
20:     <manufacturer id="HO" name="Honda" country="Japan" />
21:   </manufacturers>
22: </cars>
```

ANALYSIS Listing 8.9 contains two major sets of data. One set is formed by `model` elements, grouped by the `models` element on line 3. The other data set is a set of `manufacturer` elements, grouped by the `manufacturers` element on line 15. Although the `models` and `manufacturers` elements are related, they have a sibling relationship, not a parent-descendant relationship. If you iterate through one of the datasets and, within that, iterate through related data from the other set, getting to the data from the outer iteration is tricky at best. The more iterations that are nested, the harder it becomes to get to the data. However, if you store information from the current iteration in a variable and then start a new iteration nested within the current one, you can access the data from the outer iteration by using the variable that was created.

For example, this situation might happen when you want to make a list of manufacturers, with each manufacturer displaying the cars that it produces. The outer iteration then loops through the manufacturers, with the inner iteration going through the cars belonging to the current manufacturer. Listing 8.10 does exactly that.

LISTING 8.10 Stylesheet with a Nested Iteration

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates select="/cars/manufacturers" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:variable name="mfc" select="." />
      <xsl:for-each select="/cars/models/model[@manufacturer = $mfc/@id]">
        <ul>
          <li>
            <xsl:value-of select="@name" />
            <xsl:text> (</xsl:text>
```

LISTING 8.10 Continued

```
26:             <xsl:value-of select="@year" />
27:             <xsl:text>)</xsl:text>
28:           </li>
29:         </ul>
30:       </xsl:for-each>
31:     </xsl:for-each>
32:   </xsl:template>
33: </xsl:stylesheet>
```

ANALYSIS In Listing 8.10, the template matching the `manufacturers` element beginning on line 16 is the place where all the action is. On line 17 an `xsl:for-each` element is used to iterate through all the `manufacturer` elements. That is the outer iteration. Line 18 writes the name of the manufacturer, and line 19 writes the country it is from. On line 20, a variable named `mfc` (short for manufacturer) is created. The value of this variable is set to the current `manufacturer` element. With each outer iteration, the value of the variable is changed to the current `manufacturer` element and is then used on line 21 to select only those cars that have the same manufacturer as the current manufacturer. The result from applying Listing 8.10 to Listing 8.9 is shown in Listing 8.11.

Note

In Listing 8.10, variables are not strictly needed. Instead of the variable, you could use the `current()` function in the select expression on line 21. However, when the nesting is deeper, you need to use variables to get to data from higher nestings.

OUTPUT **LISTING 8.11** Result from Applying Listing 8.10 to Listing 8.9

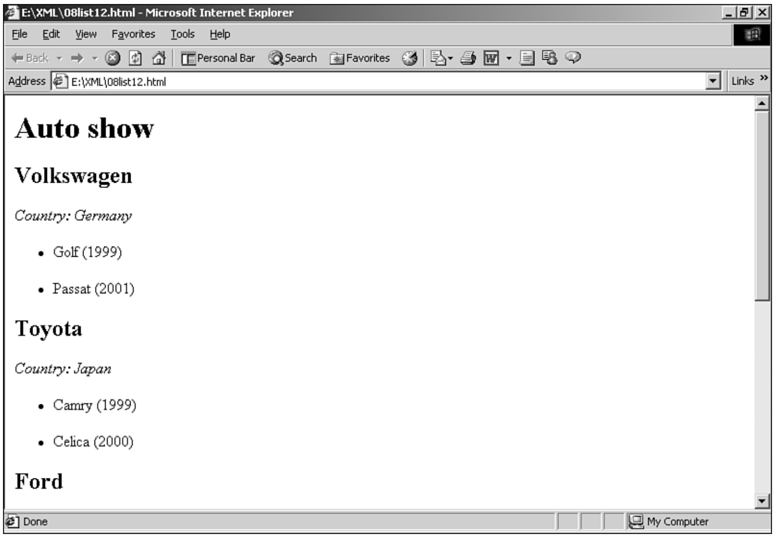
```
<html>
   <body>
      <h1>Auto show</h1>
      <h2>Volkswagen</h2>
      <p><i>Country: Germany</i></p>
      <ul>
         <li>Golf (1999)</li>
      </ul>
      <ul>
         <li>Passat (2001)</li>
      </ul>
      <h2>Toyota</h2>
      <p><i>Country: Japan</i></p>
      <ul>
         <li>Camry (1999)</li>
      </ul>
      <ul>
```

```
            <li>Celica (2000)</li>
         </ul>
         <h2>Ford</h2>
         <p><i>Country: USA</i></p>
         <ul>
            <li>Focus (2000)</li>
         </ul>
         <ul>
            <li>Mustang (2001)</li>
         </ul>
         <h2>Chevrolet</h2>
         <p><i>Country: USA</i></p>
         <ul>
            <li>Prizm (2000)</li>
         </ul>
         <ul>
            <li>Corvette (2002)</li>
         </ul>
         <h2>Honda</h2>
         <p><i>Country: Japan</i></p>
         <ul>
            <li>Civic (2000)</li>
         </ul>
         <ul>
            <li>Accord (2002)</li>
         </ul>
      </body>
   </html>
```

When you view Listing 8.11 in a browser, the result looks like Figure 8.2.

FIGURE 8.2

The result in Listing 8.11 when viewed in a browser.


E:\XML\08list12.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address E:\XML\08list12.html
Links
Done
My Computer

Auto show

Volkswagen

Country: Germany

- Golf (1999)
- Passat (2001)

Toyota

Country: Japan

- Camry (1999)
- Celica (2000)

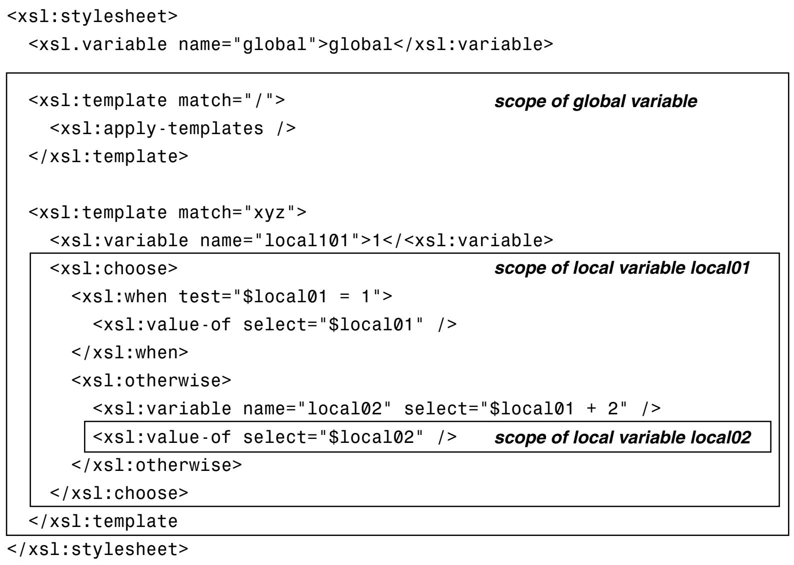
Ford

The *scope* of a variable is the area in which it is available. Outside that scope, the variable does not exist.

The boundaries of the scope are determined by the parent element and the position of the `xsl:variable` element. The variable is available only within the parent element, which means that it is available only to its sibling elements and their descendants. Also, it is available to siblings only after the variable's definition. In other words, the variable's scope is the *following* axis. With a template or with an `xsl:for-each` iteration, the variable created inside the template body (or the body of the `xsl:for-each` element) is re-created each time the template is matched or a new iteration is started. So, the variable in Listing 8.10 doesn't really change; it is actually re-created for each iteration.

The scope of a variable is not limited to templates and iterations. A variable declared within the body of an `xsl:if` element is available only to the following axis within the `xsl:if` element, not outside it. Variables declared in different parts of the stylesheet can therefore have the same name, as long as their scopes do not overlap. When the scope of two variables overlaps and they have the same name, the variable created in the broadest scope is not accessible. Say you create a global variable and a variable within a certain template that has the same name. Within that template, only the variable created there is available; the global variable is not. Figure 8.3 shows several variable declarations and shows you where the declared variables are in scope.

FIGURE 8.3
Scope of different variables.



```
<xsl:stylesheet>
  <xsl.variable name="global">global</xsl:variable>

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

  <xsl:template match="xyz">
    <xsl:variable name="local101">1</<xsl:variable>
    <xsl:choose>
      <xsl:when test="$local01 = 1">
        <xsl:value-of select="$local01" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="local02" select="$local01 + 2" />
        <xsl:value-of select="$local02" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template
</xsl:stylesheet>
```

scope of global variable
scope of local variable local01
scope of local variable local02

Note

So that you have a clear picture of the scope of the variables in Figure 8.3, superfluous elements and attributes have been removed.

Creating Variables from XSLT Elements

Instead of using the `select` attribute, you also can create variables by using XSLT elements in the body of the `xsl:variable` body element. Creating variables this way is sort of like creating XSLT output, but instead of sending the result to the output, the resulting node-set, tree fragment, or simple value is stored in the variable and is usable as long as the variable is in scope. This way, you can create variables depending on some condition or build them from one or more iterations. A simple example of such a variable is shown in Listing 8.12.

LISTING 8.12 Variable Created from XSLT Elements

```
<xsl:variable name="color">
  <xsl:choose>
    <xsl:when test="@year = '1999'">
      <xsl:value-of select="$lightblue" />
    </xsl:when>
    <xsl:when test="@year = '2000'">
      <xsl:value-of select="$lightgray" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$lightgreen" />
    </xsl:when>
  </xsl:choose>
</xsl:variable>
```

ANALYSIS Listing 8.12 yields a variable that contains the value of the variables `lightblue`, `lightgray`, or `lightgreen`. The variable's value is built from the result of the XSLT fragment. In this case, the value is simple or complex, depending on the value of the variable from which this variable will take its value. You also can create variables consisting of multiple nodes by using other XSLT elements.

A common mistake is to write the code in Listing 8.12 as shown in Listing 8.13.

LISTING 8.13 Erroneous Variable Created from XSLT Elements

```
<xsl:choose>
  <xsl:when test="@year = '1999'">
    <xsl:variable name="color" select="$lightblue" />
  </xsl:when>
  <xsl:when test="@year = '2000'">
```

LISTING 8.13 Continued

```
    <xsl:variable name="color" select="$lightgray" />
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="color" select="$lightgreen" />
  </xsl:when>
</xsl:choose>
```

ANALYSIS Listing 8.13 does not yield the same result as Listing 8.12. As soon as the `xsl:choose` element is closed, the variable created within its body goes out of scope and is no longer available. This means that the variable is available only inside each choice node. In contrast, the variable created in Listing 8.12 is the result of the entire `xsl:choose` element and is available to all its following siblings and their descendants, which is probably the result you wanted.

Summary

Today you learned that you can create variables for several purposes. Variables are useful as a means of storing information in a central location so that you don't have to edit values throughout a stylesheet. Instead, you can edit the values in one place, with the additional advantage that you don't forget to change any values. Creating such variables is straightforward because you can just hard-code the values into the `xsl:variable` body.

You also learned that variables can be created with a value that comes from the source XML. You can do so by using a select expression or XSLT elements in the `xsl:variable` body to construct the value. In the latter case, the value doesn't necessarily have to map one to one with a fragment of the source XML. Creating variables from the source XML can make your code easier to read because you have handy names for data coming from complex expressions or a constructed value. If you want to use data that will become hard (or impossible) to address because of nesting XSLT structures, using variables might be the only solution.

Tomorrow's lesson will discuss a different type of variable. These variables also can be used to pass on information from one template to another, or even from outside the stylesheet.

Q&A

Q Can I give a value to a variable both by using the `select` attribute and a value inside the element?

A No. If you create a variable by using a `select` attribute, the body of the `xsl:variable` element must be empty.

Q Can I use a separate source document to store all variable data, just like a CSS?

A Yes. This topic will be covered on Day 14, "Working with Multiple XML Sources."

Q Can I pass variables on to a template when using `xsl:call-templates` or `xsl:apply-templates`?

A No. Another element in XSLT makes this task possible. This topic will be discussed in tomorrow's lesson.

Q Why would I use variables instead of attribute-sets?

A Attribute-sets are a good alternative for static, hard-coded information common to several elements. Variables can contain dynamic information and can also be used in element values.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Variables need to have a unique name within a stylesheet.
2. True or False: A global variable is accessible in an entire stylesheet.
3. How do you create a variable named `car` with `Ford Focus` as its value?
4. What is the advantage of using the `xsl:variable` element body to create a variable from values in the source XML over using the `select` attribute?
5. Is it possible to create a variable that holds only an attribute with its value?

Exercise

1. Create a stylesheet that displays the cars in Listing 8.9 in a table (similar to Listing 8.3), with rows of different car years colored differently. Years 2001 and later might be colored the same.

This page intentionally left blank

DAY 9

Working with Parameters

Yesterday you learned how to use variables to make stylesheets easier to read and edit and to perform tasks that are otherwise very hard or maybe even impossible to achieve. Variables therefore are very useful, but you cannot perform some tasks with variables.

In today's lesson, you will learn about parameters. Parameters and variables are very much related, but with some key differences that are rooted in their different purposes. Unlike variables, parameters are meant to pass information from one template to another, or from outside a stylesheet into a stylesheet.

Today you will learn why these capabilities are useful and what you can do with them. Today you will learn the following:

- What parameters are
- How to create parameters
- How to change processing with a parameter
- How to create templates that return a value based on a parameter
- How to get information from outside a stylesheet into a parameter

Understanding Parameters

So far, the stylesheets and templates you have created are static. By that, I mean that they operate only on data in the source XML and create output accordingly. No data from outside the source XML and the stylesheet have impact on the way the source XML is processed. This is not a failing; you have seen that you can do amazing things with very little code. However, life would be a lot easier if you could influence the appearance of some data output by using data that is not stored in either the source XML or stylesheet. These sources are supposed to be relatively static. You don't want to change a stylesheet just because you want output to be a little different sometimes; you also don't want to create a separate stylesheet because doing so creates too many problems when you have to change some things common to both stylesheets. What you really want in such a scenario is to be able to dynamically influence the output. This is where parameters come in.

What Are Parameters?

Parameters come in two shapes: global and local. Global parameters receive data from outside the stylesheet and source XML, so you can make the stylesheet create a different output. Local parameters are similar but are used within a template to receive data from the template that instantiated it, either by calling or matching the template. Based on the data received by the parameter or parameters, the template can adjust its processing and create different output. This is a one-way process because a template adjusts its processing to create different output; it does not change the value of a parameter and send it back. You can, however, capture the output of a template in a variable, so the template acts more or less like a function.

A parameter is a receiver. You also need a sender for the parameter to work. For a global parameter, the sender is the processor itself. The processor must have some mechanism that allows you to send data into the stylesheet. For local parameters, `xsl:call-template` and `xsl:apply-templates` can be fitted with a child element that sends data to a parameter in the called or match template.

A parameter itself is much like a variable. In fact, after it is created, it feels and acts just like a variable—so much so that you address a parameter as if it were a variable. The same rules with regard to naming, scope, and uniqueness apply to both variables and parameters. If you define a parameter with the same name as a variable, they are both treated as if they were variables. A parameter, as such, can contain the same types of values, created in much the same way. Creating a parameter is like creating a variable, except for the element name, of course. The code creating a parameter therefore can be one of the following:

```
<xsl:param name="$carmodels" select="cars" />
```

```
<xsl:param name="$carmodels"><xsl:value-of select="cars"></xsl:param>
```

Creating a parameter this way is identical to creating a variable. As with variables, you can use the preceding methods to assign both simple and complex values to parameters. A key difference between parameters and variables is that a parameter may be created only as a top-level element or as an immediate child of a template. In addition, if a parameter is created in a template, it needs to precede any other elements so that all parameters are known when template processing starts.

The value of a parameter is determined from the `select` attribute or the element body if no `select` attribute exists. This is the same as with a variable. However, these values are used only if no value is received for the parameter. So, the value given by the `select` attribute or the element body is actually a default value to be used when no value is received.

Because parameters are named, you can send specific data to a specific parameter. If you don't send data to a parameter, it uses its default value. Conversely, if you send data to a parameter that doesn't exist, it is ignored. One reason it does not create an error is that when you are using `xsl:apply-templates`, different templates can match, some of which do not need or have certain parameters.

If you have a template with a parameter as defined in the preceding paragraph, you can call and send a value to it by using the code in Listing 9.1.

LISTING 9.1 Template Call with a Parameter

```
<xsl:call-template name="models">
  <xsl:with-param name="carmodels" select="models" />
</xsl:call-template>
```

ANALYSIS The `xsl:with-param` element on the second line defines the value that is sent to the parameter `carmodels` in the template being called. If you use template matching instead of calling, you can just replace `xsl:call-template` with `xsl:apply-templates`, and you're in business. The `xsl:with-param` element works exactly the same as `xsl:variable` and `xsl:param` to acquire the value sent to the template.

What Is the Benefit of Parameters?

Parameters offer great benefits, both to make some common tasks easier and to add functionality. You will find that many of the tasks you can perform using parameters are not possible without them.

With information coming from outside a template or stylesheet, you can adapt your stylesheet so that it can perform tasks a little differently if a parameter has a certain value. This way, you can generate output that is slightly different just because a parameter's value is different. This capability is particularly useful if you have an element that can have different parent elements, and the output needs to be slightly different for the different parent elements. You also can achieve this result by using `xsl:for-each` within the template matching the parent element, but doing so would mean sacrificing your generic approach. In general, parameters enable you to make generic templates and stylesheets that will be useful in many situations. Because you often will output the same elements more or less the same, using parameters this way is much more efficient than creating additional templates or even additional stylesheets.

You also can create templates that operate on only the parameter or parameters sent. You then can let the template create output, but you can also capture the output from the template in a variable. If you do that, the template is more or less like a function returning a value.

Finally, you can use parameters to add data to an XML source programmatically. This means, of course, that you need to receive data from outside the stylesheet. One common scenario in which this could happen is in a Web-based environment where a form is posted to a server and the values in the form need to be added to an XML source in one form or another. Bonus Project 3, "Creating a Shopping Basket in XSLT," at the end of Week 3 uses this concept to alter the contents of a user's shopping basket.

Using Parameters

Using parameters is relatively painless. As I said before, creating parameters is just like creating variables, which you already know how to do. Sending a value to a parameter from within another template is also similar to creating a variable, so I will concentrate on functionality rather than form.

Using Parameters to Alter the Output

You have already seen some methods to alter output according to element or attribute values, the position of a node in a node-set, and other data. Be ready to learn another method because this one is quite important. Using parameters, you can greatly reduce the amount of code you have to write to get something done, especially when output becomes more complex. The sample used here is similar to a sample used in yesterday's lesson. Again, the source XML is the familiar list of cars, as shown in Listing 9.2.

LISTING 9.2 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

Yesterday several samples operating on the XML source in Listing 9.2 created HTML showing data in a table with alternating row colors: a gray background and a white table background. The stylesheet in Listing 9.3 creates the exact same output but is changed to use parameters.

LISTING 9.3 Stylesheet Creating HTML Table from Listing 9.2

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />

  <xsl:variable name="bgcolor">#cccccc</xsl:variable>
  <xsl:variable name="altbgcolor">#ffffff</xsl:variable>

  <xsl:template match="/">
    <html>
    <body bgcolor="{$bgcolor}">
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="cars">
    <table bgcolor="{$altbgcolor}" width="75%">
      <xsl:for-each select="car">
        <xsl:choose>
          <xsl:when test="position() mod 2 = 0">
            <xsl:call-template name="car">
              <xsl:with-param name="carbgcolor" select="$altbgcolor" />
            </xsl:call-template>
          </xsl:when>
          <xsl:when test="position() mod 2 = 1">
            <xsl:call-template name="car">
              <xsl:with-param name="carbgcolor" select="$bgcolor" />
            </xsl:call-template>
```

Listing 9.3 Continued

```
31:           </xsl:when>
32:           <xsl:otherwise>
33:             <xsl:call-template name="car" />
34:           </xsl:otherwise>
35:         </xsl:choose>
36:       </xsl:for-each>
37:     </table>
38:   </xsl:template>
39:
40:   <xsl:template name="car">
41:     <xsl:param name="carbgcolor">#ffffff</xsl:param>
42:     <tr bgcolor="{$carbgcolor}">
43:       <td><xsl:value-of select="@model" /></td>
44:       <td><xsl:value-of select="@manufacturer" /></td>
45:       <td><xsl:value-of select="@year" /></td>
46:     </tr>
47:   </xsl:template>
48: </xsl:stylesheet>
```

Analysis Listing 9.3 has several changes with regard to Listing 8.4 used in yesterday's lesson. The easiest one to spot is the difference in the template starting on line 40. In Listing 8.4, that template added only three table cells, one for each attribute. It did not, however, add a table row because that task was performed in the template that calls this template. The `tr` element had to be created with a dynamic attribute. The value of the attribute was determined with a complex expression and an `xsl:choose` construction. Although that approach worked, creating the table rows in the template that deals with each `car` element is much more logical. In general, it is always a good idea to keep related elements together in XSLT and create output from a specific element in one place, not multiple places.

The template on line 40 starts with a parameter declaration on line 41. A parameter is defined with the name `carbgcolor`, which gets the default value `#ffffff` (which is white). This optional default value is used if no value is passed from the calling template. You also can leave this value empty and just define the parameter's name. If you use this approach and no value is passed on for that parameter, its value is an empty string. The parameter is used when the `tr` element is created as the value of the `bgcolor` attribute. Here, you just have to use curly braces surrounding the value instead of creating an `xsl:attribute` element, as in Listing 8.4, because here no choices are made; the choices are still made in the other template.

Two other changes have been made to Listing 9.3 compared to Listing 8.4, all in the template starting on line 18, matching the `cars` element. First, the `tr` element that was created in this template has been removed because this element is now created in the template on line 40. Additionally, the `xsl:call-template` elements on lines 23 and 28 contain `xsl:with-param` elements that specify the values to be given to the `carbgcolor` parameter in the template being called.

If you compare Listing 8.4 to Listing 9.3, you will notice that Listing 9.3 is a few lines longer than Listing 8.4. So, why create the stylesheet this way? The foremost reason is logical grouping. The more logical a stylesheet is, the easier it is to read and maintain. The template on line 40 creates a table row with a flexible background color, plain and simple. You can easily imagine what the output will look like. In Listing 8.4, the creation of the table row was broken up into separate pieces. The `xsl:attribute` and `xsl:choose` element in between also make it difficult to imagine what the output will be. Using parameters, you can keep the complex logic together and make templates that create very clean output. In Listing 9.4, the template matching the `cars` element is separated into two templates to create an even cleaner look.

LISTING 9.4 Partial Stylesheet Simplified for Reading

```
1:  <xsl:template match="cars">
2:    <table bgcolor="{$altbgcolor}" width="75%">
3:      <xsl:call-template name="carrows" />
4:    </table>
5:  </xsl:template>
6:
7:  <xsl:template name="carrows">
8:    <xsl:for-each select="car">
9:      <xsl:choose>
10:       <xsl:when test="position() mod 2 = 0">
11:         <xsl:call-template name="car">
12:           <xsl:with-param name="carbgcolor" select="$altbgcolor" />
13:         </xsl:call-template>
14:       </xsl:when>
15:       <xsl:when test="position() mod 2 = 1">
16:         <xsl:call-template name="car">
17:           <xsl:with-param name="carbgcolor" select="$bgcolor" />
18:         </xsl:call-template>
19:       </xsl:when>
20:       <xsl:otherwise>
21:         <xsl:call-template name="car" />
22:       </xsl:otherwise>
23:     </xsl:choose>
24:    </xsl:for-each>
25:  </xsl:template>
```

ANALYSIS Listing 9.4 creates the exact same output as Listing 9.3. Look at the template on line 1 matching the `cars` element. Instead of having a lot of processing inside the `table` element, a single template call now appears on line 3, calling a new template named `carrows`. The new template starts on line 7 and produces no output whatsoever. It iterates through all the `car` elements and determines the background color to be used for the table row. When it's time to create output, this template calls the template that outputs the table row, which is on line 40 of Listing 9.3. The name of the new template tells you exactly what it does, keeping the template matching the `cars` element nice and clean. All the nasty, hard-to-read stuff is in one template, but all templates that create output are nice and simple. Creating stylesheets this way is particularly handy if you work with several people on a project. Chances are, you have a person aboard who knows a lot about HTML but not about XSLT. This person probably knows his or her way around the pieces of the stylesheet that matter to him or her; those pieces contain mostly HTML.

Apart from making stylesheets easier to read, using parameters also is more efficient when stylesheets become more complex. The preceding samples are relatively simple, making the parameters extra overhead, taking more lines of code. When code becomes more complex, the overhead becomes smaller and eventually will become more efficient.

Using Parameters to Create Template Functions

Using templates is a great way to break up functionality into smaller, manageable pieces. This is true for both match templates and called templates. Normally, the result of a template is written to the output, but you also can capture the output in a variable, by matching or calling templates from within the body of an `xsl:variable` element. This way, you can operate on that data in a later stage of processing. Capturing the data returned from a template in a variable transforms that template into a function returning a value. Using parameters, you can increase the functionality of such functions because you can provide additional data that may affect how the current data is processed. You may even create a template that will operate only on the data provided through a parameter, which allows you to operate on data that is not in context.

Yesterday's lesson contained a sample creating a week menu from a data file with menu data. That sample, Listing 8.7, is repeated in Listing 9.5.

LISTING 9.5 Sample XML Source with Menu Data

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
```

```
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolate Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

In yesterday's lesson, Listing 8.6 produced a nice menu of the week, using variables to get the separate dishes. Listing 9.6 does exactly the same but is a complete revamp of Listing 8.6. You won't recognize much because the approach taken is completely different, using a template function.

LISTING 9.6 Stylesheet With Function Template

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:variable name="weekmenu">
      <xsl:call-template name="getmenu">
        <xsl:with-param name="week">7</xsl:with-param>
      </xsl:call-template>
    </xsl:variable>
    <xsl:call-template name="displaymenu">
      <xsl:with-param name="menu" select="$weekmenu" />
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="getmenu">
    <xsl:param name="week">1</xsl:param>
    <xsl:copy-of select="/menu/appetizers/dish[position() =
       ((($week - 1) mod count(/menu/appetizers/dish)) + 1)]" />
    <xsl:copy-of select="/menu/entrees/dish[position() =
       ((($week - 1) mod count(/menu/entrees/dish)) + 1)]" />
    <xsl:copy-of select="/menu/desserts/dish[position() =
       ((($week - 1) mod count(/menu/desserts/dish)) + 1)]" />
  </xsl:template>

  <xsl:template name="displaymenu">
    <xsl:param name="menu" />
```

LISTING 9.6 Continued

```
30:     <xsl:if test="$menu">
31:       <xsl:text>This week's menu:</xsl:text>
32:       <xsl:for-each select="$menu/dish">
33:         <xsl:text>&#xA;- </xsl:text><xsl:value-of select="." />
34:         <xsl:text> $</xsl:text><xsl:value-of select="@price" />
35:       </xsl:for-each>
36:     </xsl:if>
37:   </xsl:template>
38: </xsl:stylesheet>
```

ANALYSIS Unlike Listing 8.6, Listing 9.6 does not use any global variables. Also, whereas Listing 8.6 used one root template, Listing 9.6 employs three templates. The root template on line 7 calls the other two templates to do the work. The template on line 18, named `getmenu`, determines which dishes are to be added to this week's menu, based on a parameter named `week`. This template acts as a function in that it returns a node-set with the dishes that are chosen for the menu of the week. The expressions used to select the elements to be part of the node-set are the same as those used in Listing 8.6. The `week` parameter defaults to week 1 if no value is passed on, so you will always get a menu of the week, even if the parameter isn't given a value from outside the template. But in Listing 9.6, line 10 gives the parameter the value `7`. Instead of creating output, the `getmenu` template copies the `dish` elements from `appetizers`, `entrees`, and `desserts`, including their attributes. Normally, that would mean that these elements would be copied to the output just as they appear in the source XML, but in Listing 9.6, the output is captured in a variable named `weekmenu` on line 8. Note that the elements copied are selected based on the same complex expressions that were used to create the global variables in Listing 8.6.

On line 13, the root template calls a second template, named `displaymenu`, which does what its name suggests. On line 14, the menu assembled with the `getmenu` template is passed to the `displaymenu` template as the parameter `menu`. The `menu` parameter, defined on line 29, defaults to an empty parameter. Line 30 actually tests whether the parameter is empty. Only if it is not empty should any output be created.

One big difference between Listing 8.6 and Listing 9.6 is the way the output is created. In Listing 8.6, this was done for each variable explicitly. In this case, the `getmenu` template copies the `dish` elements, so the `menu` parameter actually contains three `dish` elements. They can be processed exactly the same, as is reflected by the value of the `select` attribute of the `xsl:for-each` element on line 32. For easy reference, Listing 9.7 shows the output created when Listing 9.6 is applied to Listing 9.5.

OUTPUT **LISTING 9.7** Result from Applying Listing 9.6 to Listing 9.5

```
This week's menu:
- Smoked Salmon and Avocado Quesadilla $10.95
- Seafood Pasta $17.95
- Dame Blanche $6.95
```

Just as the previous sample, Listing 9.6 is longer than its counterpart from yesterday's lesson. Listing 9.6, however, is much more generic. If you want to create a four-course menu with two appetizers, you have to change only the `getmenu` template. The rest of the stylesheet stays exactly the same. In Listing 8.6, you were required to add a variable and add code to display that variable. This doesn't mean that you should always use parameters to solve such problems. However, you should look long and hard at the needs of the stylesheet you're going to create. If the stylesheet is likely to change regularly, using more generic templates is a good idea because they are reusable.

Expanding Functions in XSLT

The base set of functions in XSLT is quite limited. Obviously, XSLT's designers felt that you needed only basic building blocks, so you can create some common functions yourself. Because the functions you can create yourself are actually templates, unfortunately you cannot use them in expressions. You can, however, capture their output in a variable that is usable in a subsequent expression.

A function common to many programming languages is `max`, which returns the value of the largest of two numbers it has been given. In a conventional programming language,

```
max(3, 5)
```

would return 5. Such a function does not exist in XSLT, but you can easily create one. The stylesheet in Listing 9.8 contains a template that will perform this function, along with a template using it.

LISTING 9.8 Template Performing the `max` Function

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:variable name="maximum">
      <xsl:call-template name="max">
        <xsl:with-param name="x">3</xsl:with-param>
        <xsl:with-param name="y">5</xsl:with-param>
```

Listing 9.8 Continued

```
12:       </xsl:call-template>
13:     </xsl:variable>
14:     <xsl:value-of select="$maximum" />
15:   </xsl:template>
16:
17:   <xsl:template name="max">
18:     <xsl:param name="x" />
19:     <xsl:param name="y" />
20:
21:     <xsl:choose>
22:       <xsl:when test="$x > $y">
23:         <xsl:value-of select="$x" />
24:       </xsl:when>
25:       <xsl:otherwise>
26:         <xsl:value-of select="$y" />
27:       </xsl:otherwise>
28:     </xsl:choose>
29:   </xsl:template>
30: </xsl:stylesheet>
```

Analysis When you employ Listing 9.8, any source XML will suffice; it will not affect the output. The `max` template on line 17 serves as the `max` function. The value it returns is captured by the `maximum` variable on line 8, which is written to the output on line 14. When the template is called on line 9, you need to specify the value of both parameter `x` and parameter `y`. Given the two parameters, the template will return whichever value is larger. Note that when the values are equal, it doesn't matter which value is returned, so `y` is returned.

Getting Data from Outside the Stylesheet

As I said earlier, parameters are not restricted to use within templates. You also can define parameters as top-level elements so that they become global parameters. Global parameters are useful to get data from outside the stylesheet, which is not part of the XML source. You can use this approach to manipulate the output of the documents, as shown in the preceding sections, but also to add data to an XML source.

Getting Different Output

In Listing 9.6, alternating the "menu of the week is based on the value of `week` parameter." The trouble is that you still have to go into the stylesheet and change the value of the parameter to get a different menu. All that is about to change because you will be able to create the stylesheet and keep it as is. When you transform the source XML with the stylesheet using a processor, you can provide a parameter when you call the processor, so the output will change. Listing 9.9 is the first step in getting this method to work.

Listing 9.9 Stylesheet Creating a Menu of the Week

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:param name="week">1</xsl:param>
8:
9:    <xsl:template match="/">
10:     <xsl:variable name="weekmenu">
11:       <xsl:call-template name="getmenu" />
12:     </xsl:variable>
13:     <xsl:call-template name="displaymenu">
14:       <xsl:with-param name="menu" select="$weekmenu" />
15:     </xsl:call-template>
16:   </xsl:template>
17:
18:   <xsl:template name="getmenu">
19:     <xsl:copy-of select="/menu/appetizers/dish[position() =
20:         ((($week - 1) mod count(/menu/appetizers/dish)) + 1)]" />
21:     <xsl:copy-of select="/menu/entrees/dish[position() =
22:         ((($week - 1) mod count(/menu/entrees/dish)) + 1)]" />
23:     <xsl:copy-of select="/menu/desserts/dish[position() =
24:         ((($week - 1) mod count(/menu/desserts/dish)) + 1)]" />
25:   </xsl:template>
26:
27:   <xsl:template name="displaymenu">
28:     <xsl:param name="menu" />
29:     <xsl:if test="$menu">
30:       <xsl:text>This week's menu:</xsl:text>
31:       <xsl:for-each select="$menu/dish">
32:         <xsl:text>&#xA;- </xsl:text><xsl:value-of select="." />
33:         <xsl:text> $</xsl:text><xsl:value-of select="@price" />
34:       </xsl:for-each>
35:     </xsl:if>
36:   </xsl:template>
37: </xsl:stylesheet>
```

Analysis Listing 9.9 contains only a few changes over Listing 9.6. First, on line 7 a global parameter is created with a default value of `1`. So, if no value for the parameter is given, the stylesheet outputs the menu for week 1. This parameter is available in all templates, so it doesn't need to be passed on from one template to another. This means that when the `getmenu` template is called on line 11, no value has to be given for a parameter. Also, the `getmenu` template on line 18 no longer needs to have a parameter itself because it can just get the value from the global parameter, just like it would with a global variable. Other than that, the stylesheet is the same as Listing 9.6.

When you apply Listing 9.9 to Listing 9.5, you need to specify a value for the `week` parameter. How you do so is different for the different processors. With Saxon, you can add a parameter to the command line by name and assign a value to it, as follows:

```
saxon 09list05.xml 09list09.xsl week=7
```

If you run the preceding command at the command line, you get the same output shown in Listing 9.7. Without changing the stylesheet, you now can get a different week's menu just by changing the value assigned to the `week` parameter. So, for week 24, you enter the following at the command line:

```
saxon 09list05.xml 09list09.xsl week=24
```

The preceding command yields the output in Listing 9.10.

OUTPUT **LISTING 9.10** Result from Applying Listing 9.9 to Listing 9.5 for Week 24

```
This week's menu:
- Caesar Salad $6.95
- Rack of Lamb $18.95
- Banana Split $6.95
```

As you can see, passing parameters from outside the stylesheet is easy and handy. It enables you to alter the output just by changing the parameters you use when calling the processor.

Apart from Saxon, most other processors offer similar functionality to give parameter values. MSXSL uses the same syntax as Saxon, so you can run MSXSL from the command line like this:

```
msxsl 09list05.xml 09list09.xsl week=7
```

Xalan has another syntax entirely. If you want to run Xalan from the command line and insert parameters, you have to enter the following:

```
java org.apache.xalan.xslt.Process -in 09list05.xml -xsl 09list09.xsl
➥-param week 7
```

Using Parameters to Add Data

Apart from manipulating output, you also can use parameters to add data to an existing XML source. In this case, a parameter receives data from the outside world. This data then can be used to create additional elements or attributes. All you need to do is determine where any nodes should be inserted and create a template that can be called at that particular point. Listing 9.11 shows a stylesheet that adds a `car` element to the sample XML in Listing 9.2.

LISTING 9.11 Stylesheet Adding a Car Element to Listing 9.2

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:output method="xml" version="1.0" encoding="UTF-8" />
 6:
 7:   <xsl:param name="model" />
 8:   <xsl:param name="manufacturer" />
 9:   <xsl:param name="year" />
10:
11:   <xsl:template match="/">
12:     <xsl:apply-templates />
13:   </xsl:template>
14:
15:   <xsl:template match="cars">
16:     <xsl:copy>
17:       <xsl:apply-templates />
18:       <xsl:call-template name="insertcar" />
19:     </xsl:copy>
20:   </xsl:template>
21:
22:   <xsl:template match="car">
23:     <xsl:copy-of select="." />
24:   </xsl:template>
25:
26:   <xsl:template name="insertcar">
27:     <car model="{$model}" manufacturer="{$manufacturer}" year="{$year}" />
28:   </xsl:template>
29: </xsl:stylesheet>
```

ANALYSIS Listing 9.11 is basically very simple. The functionality copying nodes from the source XML were discussed on Day 5, "Inserting Text and Elements." However, some sections are important here. First, three global parameters named `model`, `manufacturer`, and `year` are defined on lines 7–9. The names of the parameters are not surprising; they correspond to the attributes of the `car` elements in the source XML. After all the `car` elements are copied over to the output, the `insertcar` template is called on line 18. On line 26, this template adds a new `car` element to the output before the `cars` element is closed.

If no parameters exist, an empty element is inserted. But, of course, you are going to insert parameters. With Saxon, you use the following command line:

```
saxon 09list02.xml 09list11.xsl model=X-type manufacturer=Jaguar year=2001
```

A previous sample used only one parameter, but here you can see that using multiple parameters is equally possible. In this case, the parameters add a Jaguar X-type. Running Saxon with the preceding parameters results in the output shown in Listing 9.12.

OUTPUT **LISTING 9.12** Result from Running Saxon to Add Data

```
<?xml version="1.0" encoding="UTF-8"?><cars>
  <car model="Focus" manufacturer="Ford" year="2000"/>
  <car model="Golf" manufacturer="Volkswagen" year="1999"/>
  <car model="Camry" manufacturer="Toyota" year="1999"/>
  <car model="Civic" manufacturer="Honda" year="2000"/>
  <car model="Prizm" manufacturer="Chevrolet" year="2000"/>
<car model="X-type" manufacturer="Jaguar" year="2001"/></cars>
```

MSXSL is the same as Saxon, but Xalan uses a different syntax, so looking at it again is useful. For Xalan, you enter the command line with parameters like this:

```
java org.apache.xalan.xslt.Process -in 09list02.xml -xsl 09list09.xsl
➥-param model X-type -param manufacturer Jaguar -param year 2001
```

Here, another difference with Xalan comes to light. The output differs slightly from that created with Saxon or MSXSL. If you execute the preceding command, the output does not look like Listing 9.12 but looks like Listing 9.13 instead.

OUTPUT **LISTING 9.13** Result from Adding Data with Xalan

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000"/>
  <car model="Golf" manufacturer="Volkswagen" year="1999"/>
  <car model="Camry" manufacturer="Toyota" year="1999"/>
  <car model="Civic" manufacturer="Honda" year="2000"/>
  <car model="Prizm" manufacturer="Chevrolet" year="2000"/>
<car year="2001" manufacturer="Jaguar" model="X-type"/></cars>
```

ANALYSIS In Listing 9.13, the attributes of the added `car` element are in reverse order. This is a perfect example of an area where the XSLT specification leaves room for interpretation. A processor doesn't have to output attributes in the order they were added. In fact, as far as a processor is concerned, the attribute order is not significant. Most processors output it as you expect, but there is the odd one out that doesn't, like Xalan.

Setting Parameters from Code

You are not likely to use parameters to get data into a stylesheet from the command prompt. In most cases, this occurs in some kind of application using XSLT to operate on and manipulate XML documents. If you're going to create such an application yourself, you need knowledge of the processor's Application Programmer Interface (API) and the environment you're running in. Saxon and Xalan both run under Java (although Saxon.exe runs under Windows), so you need to know the Java programming language.

In addition, you need to know the API of Xalan or Saxon, which have quite a few similarities. If you're running MSXML, you need to use any language on the Windows platform that is capable of calling COM components, such as Visual C++, Visual Basic, or Delphi. You also can call MSXML from a script running in the Windows Scripting Host (WSH), Internet Explorer, or in Active Server Pages (ASP).

It is beyond the scope of this book to provide elaborate samples of all the methods to call a processor and pass parameters into the stylesheet. To give you an idea of how this process typically works, Listing 9.14 shows a piece of VBScript that could be used with the sample in Listing 9.11. I chose VBScript because it is easy to read and because it works in WSH, Internet Explorer, and ASP, with minor changes.

LISTING 9.14 VBScript Function Calling MSXML with Parameters

```
Function TransformWithParam(sXML, sXSL, sModel, sMFC, sYear)
  'Declare variables
  Dim oXML, oXSL, oOutput
  Dim oTemplate, oProc

  'Create COM objects
  Set oXML = CreateObject("MSXML2.DOMDocument.3.0")
  Set oOutput = CreateObject("MSXML2.DOMDocument.3.0")
  Set oXSL = CreateObject("MSXML2.FreeThreadedDOMDocument.3.0")
  Set oTemplate = CreateObject("MSXML2.XSLTemplate")

  'Set loading properties
  oXML.async = False
  oXSL.async = False

  'Load XML and XSLT
  Call oXML.load(sXML)
  Call oXSL.load(sXSL)

  'Create XSLT processor
  Set oTemplate.stylesheet = oXSL
  Set oProc = oTemplate.createProcessor()

  'Set processor input and output
  oProc.input = oXML
  oProc.output = oOutput

  'Add parameters
  Call oProc.addParameter("model",sModel)
  Call oProc.addParameter("manufacturer",sMFC)
  Call oProc.addParameter("year",sYear)

  'Go!
  Call oProc.transform()

```

LISTING 9.14 Continued

```
36:   'Return output
37:   TransformWithParam = oOutput.xml
38:
39:   'Release objects
40:   Set oProc = Nothing
41:   Set oTemplate = Nothing
42:   Set oOutput = Nothing
43:   Set oXSL = Nothing
44:   Set oXML = Nothing
45: End Function
```

ANALYSIS In Listing 9.14, each line of code preceded with an apostrophe is a comment. Comments help you to read and understand the code. Creating a processor is a three-step process. First, you need to create a DOM object from the XSLT file, which is done with line 9 (creating the object) and line 18 (loading the stylesheet). Second, you need to create a template from the stylesheet, which is done with line 10 (creating the object) and line 21 (using the stylesheet). The last step is creating the processor itself, which is done with line 22. The source XML also needs to be loaded into a DOM object, which is done with lines 7 and 17. Some additional lines of code are involved with loading the XML and XSLT on lines 13–14, but they are of little consequence.

After the XML is loaded and the processor created, you can start setting up the processor. The first step is to determine where the XML has to come from and where the output has to go. The former is done on line 25, which specifies the loaded XML DOM object. Line 26 specifies that the output is sent to a new XML DOM object, which is created on line 8. The last step before processing the XML is adding the parameters. Fortunately, adding them is easy, as you can see on lines 29–31, which add the parameters for the `car` element's attributes. Finally, line 34 actually tells the processor to process the XML with all the given values. The output is stored in the `oOutput` object, and on line 37, the function return value is set to the actual XML string that results from processing the source XML. After that, the only step that remains is to properly dispose of all the objects that were created.

The function in Listing 9.14 can be called from another section of script to send it to the output or something different. If you run the file named 09list14.vbs that is part of the downloads, the output is shown in a Windows message box. You can run the file by double-clicking it in Windows Explorer. The following line of code is added to the 09list14.vbs file to get the message box:

```
MsgBox(TransformWithParam("09list02.xml", "09list11.xsl", "X-type",
➥ "Jaguar", "2001"))
```

Note

If you don't understand Listing 9.14 and what I've said about it, don't worry. In that case, this is probably not something you will be doing, but will instead be covered by a programmer working with one of the mentioned languages. Understanding this example isn't necessary if you're interested only in XSLT.

Summary

Today you learned that parameters can help you make common tasks easier and more generic, so your stylesheet becomes more flexible and will need less editing if you want to make changes to the output. With parameter values coming from outside the stylesheet and the XML source, you can alter the output without touching the stylesheet or source XML.

Because parameters are much like variables, working with them is not very hard. In fact, the way you create parameters and give them values is identical to the way you work with variables, but the additional functionality of parameters makes them even more useful than variables.

Tomorrow the focus shifts from how to manipulate data to the data itself. You will learn about the different data types available in XSLT and how you need to work with them. An important lesson you will learn is how to work on only part of the value of an element or attribute instead of the entire value. This comes in the form of different functions to manipulate strings of data.

Q&A

Q Is there a limit to the number of parameters I can use?

A Theoretically, no. You can use an unlimited number of parameters, of an unlimited size. A processor, however, may not be able to handle large-sized parameters because of memory constraints.

Q Is a parameter from outside the stylesheet limited to being a simple value?

A If you use the processor from the command line, yes. If you use the processor from code, you also can pass XML fragments into the stylesheet. Doing so may require some additional code in Java, Visual Basic, and so on.

Q Is it possible to add new parameters at runtime so that I can add additional data?

A No. Only the parameters that have been defined in a template or stylesheet can be used. Because parameters can contain node-sets and tree fragments, you can create a parameter that can be used for any additional data. You will, however, have to write elaborate code to use it.

Q Parameters seem like a good idea for my Web application—for instance, to give users the ability to change the look and feel of the application. Can I insert parameters from a form directly into a processor?

A You will have to write some code to call the processor from the Web server and pass values from a form to the stylesheet. In the future, this capability is likely to be added to some Web servers if not already present.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is very helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Parameters and variables may not have the same names.
2. True or False: Parameters cannot be used with `xsl:apply-templates`.
3. What happens if a value is passed for a parameter that does not exist in the template that is invoked?
4. Why can you give a parameter a value while its value will be passed on from outside a template or the stylesheet?
5. Can you define parameters anywhere in a stylesheet?

Exercise

1. Create a stylesheet that adds a dish to the menu XML in Listing 9.5. You need to be able to give all the attributes and determine if it is an appetizer, entree, or dessert.

DAY 10

Understanding Data Types

In the preceding lessons, you learned a great deal about selecting and manipulating elements and attributes of data. You learned how to select, display, and add data and also how to get certain information regarding the document as a whole or part thereof. The data values of elements and attributes themselves were ignored in the sense that what was in them was of no consequence. The values were just sent to the output. Whether the data represented text, dates, prices, or some other type of value wasn't important. What a value represents might be very important, however. A date, for instance, may need to be displayed differently for different countries.

Today's lesson covers the data in elements and attributes and the type of data they represent. What type of data a value represents is important if you want to manipulate the value—for instance, for calculations or for a more fine-grained control over what is done with certain data. Unfortunately, discussing data types is not possible without going into technical and theoretical details, so be prepared for some tough reading.

Today you will learn the following:

- What a data type is
- Which data types are available in XSLT
- How to compare different data with different data types
- How to convert data from one data type to another

Data Type Basics

The wonderful thing about XML is that it is all text. That's what makes it easy to process for any computer system—Windows, Unix, Linux, or otherwise. Another great thing is that you can name elements and attributes in such a way that the name tells you something about the data in the element or attribute. However, the fact that an XML document is all text also means that all values are text. Now suppose that the value of one element is `2` and the value of another element is `5`. Because these values are both text, adding them together will likely yield `25` instead of `7`. Text has entirely different characteristics as opposed to numbers, and therefore reacts differently when you manipulate it.

What Is a Data Type?

NEW TERM To help you get around the problem of all-text data, XSLT defines several data types. A *data type* defines how a data value is interpreted by the processor.

In essence, a data type tells the processor what to do with a value when a certain operation is requested on it. Type information also may be used to create an internal representation of an XML document, depending on how the data type is defined. This process is very much related to character encoding and output escaping. Whether characters are encoded in UTF-8, UTF-16, or some other encoding method can have a profound impact on what the data looks like when it is sent to the output. Unlike data types, however, character encoding doesn't have an impact on how you work with the values. In fact, the processor can't even tell you which encoding method was used when a character was stored or if it was output escaped.

Which representation is used to store data becomes significant when this information has an impact on how operations will behave when you apply them to the data. This is true for the conceptual level, where you need to think of numbers and other data types, but also on the storage level. You can store numbers in many different ways, for instance, each having different memory requirements. Most languages offer a variety of data types such as 8-bit integers, 16-bit integers, 32-bit integers, single-precision floating-point numbers and double-precision floating-point numbers.

New Term An *integer* is a whole number, without fractions; you know—the ones you count with. The terms 8-bit, 16-bit, and 32-bit represent the number of bits used to store the integer. A *floating-point* number is a number with fractions, such as 1.23 or 3.14159. The terms *single precision* and *double precision* do not refer to one or two decimals, rather to the amount of memory used and thus the range of numbers that it can hold.

As you can imagine, whether you're using 8-bit or 32-bit integers makes quite a bit of difference on the amount of memory used. The latter takes up four times the amount of memory of the former. However, 8-bit integers usually can handle only numbers between –127 and 128 or 0 and 255, whereas 32-bit integers can deal with numbers in the billions. You might wonder what happens when you add two 8-bit integers with the value 255. Obviously, the result will no longer fit in an 8-bit integer.

As you can see, data types are important to processing your data correctly, so it is important for you to know which data types XSLT has, how XSLT determines the data type, and how you can work with different data types.

Data Types in XSLT

XSLT supports only a handful of data types:

- Boolean
- String
- Number
- Node-set
- Tree fragment

You have already worked with all of them, but not explicitly, because you weren't yet aware of the existence of data types. Now that you know about the data types in XSLT, you are ready to learn how XSLT actually deals with them.

A data type is always relevant to a piece of data. This data either comes from a source document or from a parameter passed by the processor. All the data values in an XML document therefore are of some data type. As far as XML is concerned, all values are strings. XSLT sees these strings either as a string or a number.In an XML document, a value can't have the Boolean data type. Booleans are of use only in expressions. Technically, node-sets and tree fragments also don't exist within an XML source, but when you operate on nodes in XSLT and select data, you can, of course, get node-set or tree fragment values. When you go down to the single element or attribute level, however, all values are either strings or numbers.

The big question is "How does XSLT determine what the actual data type is?" In most programming languages, you define a variable of a certain type, and when you assign a value to that variable, that value is automatically assumed (or converted) to be of the variable's data type. XSLT doesn't have variables that operate that way because variables in XSLT aren't typed, and they serve only as a container for a value, no matter what the type. Instead, *values* in XSLT have a data type. This data type is determined dynamically, and XSLT is therefore sometimes called a *dynamically typed language*.

The fact that XSLT is dynamically typed means that when you perform some operation on a value, the value gets the data type that is most likely to be the one you need. For instance, if you perform a string operation on a number, the number is converted to the string data type implicitly. You need to be careful with these conversions because some rules apply when this conversion happens. These rules will be discussed later in the "Conversion Between Data Types" section. Before you get into those details, it's a good idea to look at the separate data types in XSLT.

XML Schema Data Types

If you're familiar with XML Schemas, you may be wondering about the limited data type support of XSLT. The XML Schema specification contains many different data types for different number types, date and time types, different string types, Boolean, Base64 encoded binary data, and so on. With XML Schemas, you also can construct your own data types from the existing base data types, giving you even more possibilities. So, why aren't these data types supported in XSLT?

The types in XSLT are actually inherited from XPath. Both the XPath 1.0 and XSLT 1.0 specifications are older than the XML Schema specification. So, XPath 1.0 or XSLT 1.0 couldn't have incorporated the XML Schema data types. The early drafts of XSLT 2.0 suggest that it will support XML Schema data types. The development of XSLT 2.0 is in its early stages, so XML Schema data types will not be supported for some time.

The Boolean Data Type

The Boolean data type is a bit odd. A Boolean value can have one of two values, `true` or `false`, but these values don't exist by themselves. By that, I mean that there is no value you can put in an XML source document or in a stylesheet that will represent either of these values directly. Instead, XSLT offers two functions: `true()` and `false()`. These functions can be used to compare the result of some expression with the Boolean values. This is different from other languages in which Boolean values do have representations that you can use instead of `true` or `false`. The most common is zero for `false` and non-zero for `true`.

The Boolean data type works in this odd way because it is basically only for internal use in XSLT. Its use is limited to expressions in which you need to actually compare with the actual value. Most of the time, you will have an expression that compares two values. This expression yields a Boolean value, which is used to do something or not. This comparison can basically occur in two separate scenarios:

- An expression in the `test` attribute of `xsl:if` or `xsl:choose`. This expression needs to return a Boolean value to determine whether the element body of these elements needs to be executed. For instance,

```
<xsl:if test="$id = @id">
  <xsl:value-of select="@name" />
</xsl:if>
```

- An expression in a predicate. This expression needs to return a Boolean value to determine whether the node being evaluated is to be part of the selection. For instance,

```
<xsl:value-of select="name[@id = $id]" />
```

When you compare two values, you can be faced with many surprises. They will be discussed later in the "Comparing Values" section.

The Number Data Type

A number is a number. Not much to say about it, is there? Well, actually, there is, because a number is not *just* a number. A number must conform to quite a few rules, which are necessary to define what happens in unusual cases, such as adding two 8-bit integers that yield a value higher than 255.

Fortunately, XSLT uses only one data type for numbers, instead of several. This means that there are no problems with conversion, and you don't have to learn different sets of rules. Keep in mind that the rules of the number type in XSLT aren't always straightforward, but if you get the general idea, you are already halfway home. So, don't worry if you can't understand all this information at once.

A number in XSLT is always treated as a double-precision 64-bit floating-point number. Its behavior follows ANSI/IEEE standard 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*, which is also used in Java. The range that can be represented by these numbers is huge, and very specific rules define what happens in cases in which results are outside this range. However, in ordinary calculations, getting results outside this range is nearly impossible.

The IEEE 754-1985 standard defines six different "ranges" of values: finite non-zero, positive zero, negative zero, positive infinity, negative infinity, and Not a Number (NaN). Each of them is covered in detail in the following sections.

Finite Non-Zero Values

Finite non-zero values are those values that can be expressed explicitly with the memory space available (64 bits). Under this category fall all integer numbers within a certain range and fractional numbers with quite a few numbers after the decimal point. All values that can be expressed as finite non-zero numbers in IEEE 754-1985 can be expressed with the following formula:

$$s \text{ x } m \, 2^{x}$$

In this formula `s` is the sign (negative or positive), `m` the mantissa (a number between 1 and 2^{53}-1, inclusive), and `x` the exponent (which needs to be a number between -1,075 and 970, inclusive). Remembering all these details is not very important, but this formula means that the largest integer you can represent is 8.9885E307, or 89885 with 303 zeros. The largest negative integer is, of course, the opposite: -8.9885E307.

Note

The number 8.9885E307 is written in scientific notation, where the number after the E is the exponent. To get the actual number, you have to multiply the part in front of the E by 10 to the power of the given exponent. If the exponent is positive, the decimal point is pushed backward (making the number larger) with the number given by the exponent. If the exponent is negative, the decimal point is pushed forward (making the number smaller).

Caution

XSLT does not support scientific notation, so you have to write out numbers in full, both in front of and after the decimal point. Be aware that using numbers this way can take quite a bit more space than in other formats, so an XML file consisting of mainly large numbers can be quite large.

Positive Zero

The term *positive zero* looks kind of strange. Zero is zero, isn't it? Obviously, it is not, which has to do with how very large and very small numbers are dealt with. A number is positive zero in the following cases:

- Subtracting a number from itself—for instance, `10 - 10`.
- Dividing a very small positive number by a very large positive number, or dividing a very small negative number by a very large negative number. The values have to be so small or large that the result can no longer be represented as a finite non-zero number.
- Dividing any positive number by positive infinity.

Negative Zero

Negative zero, of course, is positive zero's evil twin brother. You get this value in the following cases:

- Dividing a very small positive number by a very large negative number
- Dividing a very small negative number by a very large positive number
- Dividing any negative number by positive infinity
- Dividing any positive number by negative infinity

Negative zero cannot be written as a value directly. It can only be the result of an expression. If you need a variable to be negative zero, you can use the expression `-0` to assign it to the variable. You need this value if you are doing computations in a stylesheet that shouldn't yield negative zero. With this value, you can check whether the calculation is correct.

Caution

There is debate whether `-0` means negative zero or subtraction from zero. In the latter case, the result of this expression would actually be positive zero. Although most processors use the former meaning, a processor may use the latter, giving you an undesired result. In that case, you can get negative zero by dividing 1 by negative infinity.

Positive and negative zero are required to make a distinction between operations that yield a positive or negative number that is too small to be expressed as a finite number. In those cases, these values act as approximations for the actual result. Because they are approximations, you may want to know if the result is an approximation of a negative or a positive number.

Positive Infinity

Positive infinity is used when numbers get so large they can no longer be represented as finite non-zero numbers. This happens in the following cases:

- Adding two very large positive numbers
- Multiplying two very large numbers, both positive or both negative
- Dividing any positive number by zero

Positive infinity cannot be written as a value directly. It can only be the result of an expression. If you need a variable to be positive infinity, you can use the expression `1 div 0`, to assign it to the variable. As with negative zero, the most common use of this value is checking whether a computation is correct.

Note

Division by zero is considered an error in many programming languages. With any language conforming to the IEEE 754-1985 standard, it has a well-defined result, being positive infinity.

Negative Infinity

Where there's a positive, there is, of course, also a negative. *Negative infinity* is the result of the following calculations:

- Adding two very large negative numbers
- Multiplying two very large numbers, of which one is positive and the other negative
- Dividing any negative number by zero

Negative infinity cannot be written as a value directly. It can only be the result of an expression. If you need a variable to be negative infinity, you can use the expression `-1 div 0` to assign it to the variable. This value is also mostly used to check whether a computation is correct.

Not a Number (NaN)

With all the preceding types of values, what is the use for the *Not a Number (NaN)* value? It is used basically in two situations, one in which there is no applicable value and one in which you convert a value that cannot be evaluated as a number—for instance, the string `abc`. In most other languages, such a conversion results in an error, so you have to check whether the conversion yields a valid value. In XSLT the specification is such that a stylesheet will work as long as it and the XML source document are well formed and the XSLT syntax is correct. If conversion of an invalid string value to a number would result in an error, this aim cannot be met.

NaN cannot be written as a value directly. It can only be the result of an expression. If you need a variable to be NaN, you can use the expression `number("NaN")` to assign it to the variable. This expression is covered in more detail in the section "Conversion Between Data Types."

The String Data Type

For most programmers, the string data type is very familiar. A short description is in order, though. A string is a sequence of zero or more characters—no surprises there. Basically, these characters can be any Unicode characters, but with some exceptions.

A string written as a literal in an XPath expression needs to be surrounded by single or double quotation marks—for example, `'Michiel'` or `"Michiel"`. As you learned on Day 7, “Controlling the Output,” some characters need to be output escaped if you use a literal string (in an XPath expression). In some cases, however, you don’t need to; for instance, `"Eat At Joe's"` works fine because the double quotation marks are used as the string delimiters, so the single quotation mark can be used. The other way around also works—for instance, `'He said "hello"'`.

Unicode contains several special characters (inherited from ASCII), called control characters. These characters are not supported by XSLT in any shape or form. They are leftovers from before the pretty user interfaces and stereo surround-sound cards. Many of these characters, such as the character denoting a sound beep, have no useful purpose anymore but are still part of the character set. The control characters that are supported are tab (#x9), carriage return (#xA), and new line (#xD); any other control characters are ignored by most parsers, but if not, may be the cause for errors.

The Node-set Data Type

The node-set data type is one of the types you are most familiar with from a theoretical standpoint because it was discussed thoroughly on Day 3, “Selecting Data.” I want to touch on some points again, however, because they are significant for the understanding of node-sets.

A node-set is a set, so theoretically the nodes in a node-set aren’t in any particular order. A document or XML fragment is, however, always processed in the order in which the elements appear in the source XML, which is called *document order*. This means that, in a practical sense, a node-set is ordered in document order.

A node-set contains only the nodes that match the expression used to create the node-set. Although their descendant elements are accessible from the node-set, they do not belong to the node-set itself. This, of course, also means that if you count the number of nodes in a node-set, only the nodes in the node-set itself are counted, not their descendant elements. So, the expression `count(/)` or `count(/menu)` used with Listing 10.1 yields `1` as a result. The expression `count(/menu/entrees/dish)` yields 5. Because node-sets don’t have to consist of nodes on the same level, with the same name, or even the same type, you can use axes to get other useful node-sets. For instance, if you want to know the depth of a certain node in the source document, you can use the expression `count(ancestor-or-self::*)`. If the context node is any dish element in Listing 10.1, this expression yields 3 because the node-set consists of the `dish` element itself, the parent element, and the grandparent element `menu`.

LISTING 10.1 Sample XML

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Ceasar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

One point to keep in mind is that there is no data type to represent a single node. A single node is therefore represented by a node-set containing the one node.

The Tree Fragment Data Type

When you first learned about tree fragments, you really had no way of creating one. You could create something that felt like one, but a tree fragment in itself cannot be created from just an expression. Tree fragments can exist only inside a variable or parameter, and then only when explicitly created as such. An example of a tree fragment created in a variable is shown in Listing 10.2.

LISTING 10.2 Tree Fragment in a Variable

```
<xsl:variable name="tree">
  Canadian <athlete>sprinter</athlete> <person>John Benson</person>
  ran the <event>100 meters</event> in a new world record time of
  <time>4.98 seconds</time>.
</xsl:variable>
```

ANALYSIS You might be a little surprised that Listing 10.2 doesn't have a root node. Line 2 just starts with text, and the value contains some elements as well. The whole value of the variable is mixed content from lines 2–4. For all practical purposes, the `xsl:variable` element acts as a root node. So, the text at the start of the fragment is, in fact, just a text node that is created under the root. A pitfall here is that because the variable acts as the root node, you may think that you can add attributes without first adding an element, as shown in Listing 10.3.

LISTING 10.3 Variable with Attributes

```
1: <xsl:variable name="tree">
2:   <xsl:attribute name="title">New 100m World Record</xsl:attribute>
3:   Canadian <athlete>sprinter</athlete> <person>John Benson</person>
4:   ran the <event>100 meters</event> in a new world record time of
5:   <time>4.98 seconds</time>.
6: </xsl:variable>
```

ANALYSIS Line 2 in Listing 10.3 adds an attribute to the variable. Because there is no explicit element, this is illegal. A processor will report an error if the code in Listing 10.3 is used.

A tree fragment is useful only for storing data temporarily or passing it on to a template. When you want to display a tree fragment, you can just use `xsl:value-of`, but this approach concatenates all text in the tree fragment's elements and displays it as a string. If you want to display the text otherwise, you have to apply some form of processing—for instance, with a template.

Conversion Between Data Types

Data types are useful to determine how an operation on a data value should be interpreted and what the resulting value should be. But what if the value you want to perform an operation on doesn't have the data type you require? In that case, you can explicitly convert values to a certain data type so that you are sure that any operations performed on the value are executed correctly. In other cases, data type conversion may occur implicitly—for instance, when you compare two values of different types.

Explicit Data Type Conversion

XSLT offers three functions to explicitly convert a value from one data type into another. These functions—`boolean()`, `number()`, and `string()`—convert the value given between the parentheses to the type corresponding to the function name, as in these examples:

```
boolean(5)

number('1.23')

string(/menu/entrees)
```

Conversion from one data type to another is bound to very specific rules, so you always know what the outcome is. You can convert only to simple data types. You cannot convert a Boolean, number, or string value into a node-set or a tree fragment.

Conversion to Boolean

The rules surrounding conversion of values to the Boolean data type are relatively simple. They are shown in Table 10.1.

TABLE 10.1 Conversion to Boolean from Other Data Types

Data Type	*Result*	
Number	`0`	→ `false`
	Otherwise	→ `true`
String	Empty string	→ `false`
	Otherwise	→ `true`
Node-set	Empty node-set	→ `false`
	Otherwise	→ `true`
Tree fragment	Conversion via string (see Table 10.3)	

As you can see in Table 10.1, determining the value of a conversion to Boolean is not hard. The only exception is the tree fragment, which is converted to a string first and then converted from a string to a Boolean. Basically, this means that if at least one non-empty text element exists in the tree fragment, the result is `true`. Listing 10.4 is a sample.

LISTING 10.4 Sample XML for Data Type Conversion

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

When you use the following expressions in a stylesheet that transforms Listing 10.4, they will all yield `true`:

```
boolean(/cars/car/@year)

boolean(/cars/car/@model)

boolean(/)
```

The following examples all yield `false`:

```
boolean(0)

boolean('')

boolean(//cars[@manufacturer = 'Rolls Royce')
```

Conversion to Number

Conversion to a number is well defined when you're working with string and Boolean values. For node-sets and tree fragments, the conversion becomes somewhat more difficult because these values have to be converted to strings first, as you can see in Table 10.2.

TABLE 10.2 Conversion to Number from Other Data Types

Data Type	*Result*
Boolean	`false` → `0`
	`true` → `1`
String	Parsed as a decimal number
Node-set	Conversion via string (see Table 10.3)
Tree fragment	Conversion via string (see Table 10.3)

When a tree fragment is converted to a string, all text nodes are concatenated. If that results in a string that can be parsed as a decimal number, you're in business. Consider the variable containing a tree fragment in Listing 10.5.

LISTING 10.5 Variable Containing a Tree Fragment

```
<xsl:variable name="tree">
  123
  <decpoint>.</decpoint>
  <fraction>456</fraction>
</xsl:variable>
```

Analysis If the variable in Listing 10.5 is converted to a number, using `number(tree)`, the resulting value is `123.456`. If the tree fragment contains any character that has no place in a number, such as `'abc'`, the tree fragment yields a string that cannot be parsed as a number and therefore results in `NaN`.

Conversion to String

Conversion to string is very important because, as you learned previously, the string data type is used as an intermediate data type when converting node-sets and tree fragments to Boolean and number data types. The way these conversions behave is shown in Table 10.3.

Table 10.3 Conversion to String from Other Data Types

Data Type	*Result*
Boolean	`false` → `'false'`
	`true` → `'true'`
Number	Converted to a string in decimal number format
Node-set	String value of the first node in the node-set (in document order)
Tree fragment	Concatenates all string values in the tree fragment

Conversion to the string data type looks more complicated than it is. Boolean and number values are straightforward, but you need to be aware that if you have large number values or values with very many places after the decimal point, the text representation can easily span 200 characters. Values that are not finite have different text representations, as shown in Table 10.4.

Table 10.4 Number Conversions for Nonfinite Numbers to Strings

Number Value	*Result*
Negative and positive zero	`'0'`
Positive infinity	`'Infinity'`
Negative infinity	`'-Infinity'`
NaN	`'NaN'`

Implicit Data Type Conversion

Implicit data type conversions happen when you compare two values or write values to the output. The rules employed by implicit conversions are the same as for explicit conversions. Understanding these rules is very important because you will likely be faced with them often when you're creating expressions. Specifically, when you're comparing values, you may be faced with some surprising results.

When you write a value to the output, it is always converted into a string, and the conversion rules for strings are as discussed in the preceding sections and shown in Tables 10.3 and 10.4.

Conversion Pitfalls

Because of the way the conversion rules are set up, you need to be aware of some pitfalls. One of the most common concerns conversion from one type to another and back again. For instance, if you write

```
boolean(string(false()))
```

the result is `true` instead of `false` because the inner expression yields the string `false`, which is not empty. When you convert back to Boolean, the nonempty string is regarded as `true`. This issue also is a concern when you convert between numbers and strings. For instance,

```
number(string(-1 div 0))
```

results in a number with the value `NaN`. The inner expression `-1 div 0` results in negative infinity. However, when that result is converted to a string, the string value is `-Infinity`, which results in `NaN` when converted back to a number.

Comparing Values

Being able to compare values is one of the key operations in XSLT. It is crucial for selecting data and taking action based on certain values. When you compare two values of different types, one value is converted into another. This conversion is governed by the rules employed for explicit conversion, but other rules tell you which value is being converted and which stays the same. These rules can be summed up as follows:

- When you compare a string with a number, the string is converted to a number.
- When you compare a string with a Boolean, the string is converted to a Boolean.
- When you compare a number with a Boolean, the number is converted to a Boolean.
- When a node-set or tree fragment is compared with a simple data type, the node-set or tree fragment is converted to that simple type.
- You cannot compare a node-set with a tree fragment.

From the preceding list, you can gather that when two values of different types are compared, the most basic data type is the one that stays the same. The other is converted. The only thing you have to remember is which data type is most basic. In sequence, they are as follows, from most basic to least basic:

- Boolean
- Number
- String
- Node-set/tree fragment

You can, of course, get around these rules by converting explicitly before comparing the values. This approach, however, is not applicable in many cases. It is useful only if you have two node-sets or tree fragments that you do not want to compare as such. For instance, if you want to check whether appetizers and desserts are in the menu in Listing 10.1, you must write the expression as follows:

```
boolean(//appetizers/dish) = boolean(//desserts/dish)
```

If there are appetizers and desserts, this expression returns `true`. If you write

```
//appetizers/dish = //desserts/dish
```

the two node-sets are compared, obviously resulting in `false`.

Comparison Pitfalls

You might think that comparing numbers is straightforward because numbers are so well defined. As long as you compare finite numbers, there is no problem. They are seen as the same when their values are the same. The number definition, however, does have some strange values apart from finite numbers. Some of them behave as you would expect; others don't.

When you're dealing with `NaN` values, you should be aware that, even if values were created from the same expression, comparing them still yields `false`. So, if you have a variable `x` containing a `NaN` value, the expression `$x = $x` results in `false`.

On the other hand, when you're working with positive and negative infinity, the values are regarded as the same as long as the sign is the same, even if they result from the same expression. So, the expression `1 div 0 = 2 div 0` results in `true`.

The last special case is zero. Positive and negative zero are regarded as the same, no matter the sign. So `-0 = 0` results in `true`.

Number comparison can be quite tricky, and it doesn't end there. Empty values, in particular, can wreak havoc in your stylesheet, as you can see from the following two examples:

```
boolean(@manufacturer = 'Rolls Royce')
```

```
boolean(@manufacturer != ' Rolls Royce ')
```

In Listing 10.4, there is no manufacturer named `Rolls Royce`, so one example returns `true` and the other `false`, right? Wrong, both return `false`.

Summary

Today you were bombarded with information regarding data types. Don't worry if you didn't understand it all. When you put to practice all that you have learned in this book (and all you will learn in the coming lessons), you will be reminded of some of the things you learned here when you get results that you didn't expect. Only through practice will you truly get a feel for these sorts of issues. If you stick to the basics, though, you may get very far, so I'll recap them.

XSLT has five data types; ordered from least complex to most complex, they are

- Boolean
- Number
- String
- Node-set
- Tree fragment

When values of different types are compared or used in some other operation, the value of the most complex type is converted to the value that has the least complex type. The number type is treated as a double-precision floating-point, as defined in ANSI/IEEE standard 754-1985. Although this is a well-defined standard, some odd behaviors have their roots in how this definition works. These behaviors will be obvious only when you lurk around the edges, working with numbers that are so huge or so small that they can't be expressed as finite numbers. As soon as you go into the shadows of infinity and NaN values, you're pretty much Alice in Wonderland, so expect the unexpected.

Some odd behavior can also be seen without numbers. Double conversions and comparisons with nonexisting values also give rise to unexpected results. If you truly understand everything in this lesson, you will always be able to solve the problems quickly. Otherwise, you may have a hard time.

Note

Many forums and mailing lists can help you when intricate details are important. Somebody else may spot a nuance that you overlooked or wasn't aware of. Such resources have helped me many times. Appendix D contains a list of helpful Web sites and mailing lists.

Tomorrow you will learn more details about the practical side of data types. In tomorrow's lesson, you will learn about functions that can be used to manipulate string values. They often can be used to get around the limitations of data types.

Q&A

Q Why can't I convert a simple data type to a node-set or tree fragment?

A Converting such a data type wouldn't make much sense. The best you can do is create a single node with the value, but then it would basically remain a simple type.

Q Why would I want to convert a Boolean to a string?

A I grant that you won't do this conversion often; however, you may need it when you're creating documents that contain very specific data—for certain calculations, for instance. You also can use this conversion to store their values in a document so that you can later read them back and make a string comparison to determine their Boolean value, such as `x = 'true'`.

Q Isn't there a Null value in XSLT, like in SQL or C++?

A No. There is no need for one. NaN and empty strings are used instead. Boolean values can only exist from expressions. Because the results are well defined, the Boolean type doesn't need a Null value.

Q I use very large numbers in my source XML. Can't I write them in scientific notation?

A You could create a template to dissect and convert the values into numbers. This topic will be covered in tomorrow's lesson.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is very helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: A node-set value can be converted to a Boolean value.
2. True or False: Comparing two values always yields a Boolean value.
3. Why do you need positive zero and negative zero?
4. When positive and negative zero are compared, they are the same. Why?
5. How can you check whether a number has an NaN value?

Exercise

1. Create a stylesheet that performs conversions and double conversions (back to the original data type on the values in the following XML document). Display the data type that was specified by the element name. Bonus: Compare values such as `1 div 0` and `2 div 0`.

```
<?xml version="1.0" encoding="UTF-8"?>
<datatypes>
  <number>123</number>
  <number>0</number>
  <number>-0</number>
  <number>Infinity</number>
  <number>-Infinity</number>
  <number>NaN</number>
  <number>xyz</number>
  <number></number>
  <boolean>123</boolean>
  <boolean>0</boolean>
  <boolean>true</boolean>
  <boolean>false</boolean>
  <boolean>xyz</boolean>
  <boolean></boolean>
  <string>xyz</string>
  <string></string>
  <string>123</string>
</datatypes>
```

This page intentionally left blank

DAY 11

Working with Strings

Yesterday you learned all about data types and how they affect expressions. When you create output, XML, or HTML, you always create text. The string data type therefore is the most important data type in XSLT because you always come back to it.

In today's lesson, you will learn how to manipulate strings so that you can create the output you want from just a value—for instance, outputting a number as currency or formatting data like you want it. Unlike yesterday's theoretical lesson, this lesson is about the practical application of the available functions, or tricks if you will.

Today you will learn how to do the following:

- Add strings together
- Strip space properly
- Format strings to show dates, currencies, and more
- Check strings for certain characters
- Transform strings into other strings

Operations on Strings

XSLT offers various operations that you can perform on string values. You can do anything from concatenating strings to getting pieces of strings or transforming strings into something different. These core functions can be combined to perform tasks for which no functions exist.

Gluing Strings Together

In a sense, creating XML documents is gluing pieces of data together as text. Some of these strings are eventually written to the output as element tags or attribute names, but that doesn't negate the fact that these tags also are text, and tag and attribute names can be created just like any value. The result is output that may or may not contain values that are a combination of several other values. This idea is shown in Listing 11.2, which operates on the familiar XML source shown in Listing 11.1.

LISTING 11.1 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

Listing 11.2 Stylesheet Adding Values Together

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:param name="week">1</xsl:param>
8:
9:    <xsl:template match="/">
10:     <xsl:variable name="weekmenu">
11:       <xsl:call-template name="getmenu" />
12:     </xsl:variable>
13:     <xsl:value-of select="$weekmenu/dish[1]" />
14:     <xsl:text>, </xsl:text>
15:     <xsl:value-of select="$weekmenu/dish[2]" />
16:     <xsl:text> and </xsl:text>
17:     <xsl:value-of select="$weekmenu/dish[3]" />
18:     <xsl:text> only $27.95</xsl:text>
19:   </xsl:template>
20:
21:   <xsl:template name="getmenu">
22:     <xsl:copy-of select="/menu/appetizers/dish[position() =
23:        ((($week - 1) mod count(/menu/appetizers/dish)) + 1)]" />
24:     <xsl:copy-of select="/menu/entrees/dish[position() =
25:        ((($week - 1) mod count(/menu/entrees/dish)) + 1)]" />
26:     <xsl:copy-of select="/menu/desserts/dish[position() =
27:        ((($week - 1) mod count(/menu/desserts/dish)) + 1)]" />
28:   </xsl:template>
29: </xsl:stylesheet>
```

Analysis Listing 11.2 is a somewhat simplified version of a stylesheet used in an earlier lesson. The template starting on line 21 contains the logic to create a menu of the week. Lines 13–18 are responsible for creating the output of this stylesheet. The `xsl:value-of` and `xsl:text` elements create the text in Listing 11.3.

Output **Listing 11.3** Result from Applying Listing 11.2 to Listing 11.1

```
Crab Cakes, Grilled Salmon and Dame Blanche only $27.95
```

Analysis In Listing 11.3, the result from Listing 11.2 is one "value" that is a combination of several values and text inserted as literals. This result is not shocking in itself, but it does show you the core of what XSLT is about.

You can achieve the result in Listing 11.3 in a much different way by using a function that is meant to glue strings together—in other words, concatenating several strings. This function, called `concat()`, is used in Listing 11.4 to create the same result.

LISTING 11.4 Changes to Listing 11.2 to Use `concat()`

```
1:  <xsl:template match="/">
2:    <xsl:variable name="weekmenu">
3:      <xsl:call-template name="getmenu" />
4:    </xsl:variable>
5:    <xsl:value-of select="concat($weekmenu/dish[1],', ',
6:                                 $weekmenu/dish[2],' and ',
7:                                 $weekmenu/dish[3],' only $27.95')" />
8:  </xsl:template>
```

ANALYSIS The key difference between Listing 11.2 and Listing 11.4 is the way the value is created. In Listing 11.4, line 5 employs the `concat()` function to concatenate strings. This function takes a comma-separated list of values as arguments and glues them together in one string, no matter what the initial data type of the different arguments. The individual arguments are converted with the same rules used by the `string()` function.

You might be wondering why you would use `concat()` instead of the method used in Listing 11.2, because that method may be more clear in terms of what the output will be. That is certainly true, but you also can use the `concat()` function in expressions. If you were to use the method from Listing 11.2, you would have to create a variable and use it in the expression. Whether you use `concat()` in a situation like Listing 11.4 is up to your personal preference, but in expressions, using it is essential. In combination with other functions, it can be quite useful, as I'll discuss on Day 16, "Advanced Data Selection."

Checking for Characters in a String

Especially when you work with string values that are quite long, you might want to check if that string contains some character or sequence of characters. Checking for characters is useful in parsing scenarios to see whether a string conforms to a certain syntax. XSLT offers two functions that are helpful in such cases: `contains()` and `starts-with()`. These functions are very much related to one another but offer slightly different functionality.

Using the `contains()` Function

The `contains()` function checks whether a certain sequence of characters exists in a given string. If the given sequence exists anywhere in the given string, the function returns `true`; otherwise, it returns `false`. Apart from parsing and syntax checking, you can use this function to filter out nodes that do or do not contain certain characters or sequences of characters. Using this function gives you much more fine-grained control than matching on entire nodes or values of nodes. An example is shown in Listing 11.5.

LISTING 11.5 Fine-Grained Filtering of Values

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:output method="text" encoding="UTF-8" />
 6:   <xsl:strip-space elements="*" />
 7:
 8:   <xsl:template match="/">
 9:     <xsl:apply-templates />
10:   </xsl:template>
11:
12:   <xsl:template match="menu/*">
13:     <xsl:value-of select="concat(@title,'&#xA;')" />
14:     <xsl:apply-templates />
15:     <xsl:text>&#xA;</xsl:text>
16:   </xsl:template>
17:
18:   <xsl:template match="dish">
19:     <xsl:if test="not(contains(.,'Salmon') or
20:                   contains(.,'Sea') or
21:                   contains(.,'Crab') or
22:                   contains(.,'Prawn'))">
23:       <xsl:value-of select="." /> $<xsl:value-of select="@price" />
24:       <xsl:text>&#xA;</xsl:text>
25:     </xsl:if>
26:   </xsl:template>
27: </xsl:stylesheet>
```

ANALYSIS Listing 11.5 displays a menu based on Listing 11.1. The template on line 12 matches any of the `appetizers`, `entrees`, or `desserts` elements, with line 13 inserting its title and a linefeed. Note that the linefeed is inserted using the `&#xA;` character reference. The template on line 18 processes each `dish` element, but not all `dish` elements are sent to the output. This is caused by the `xsl:if` element starting on line 19. The test expression used here makes sure that lines 23 and 24 are executed only when the string value of the `dish` element does not contain `'Salmon'`, `'Sea'`, `'Crab'`, or `'Prawn'`. In essence, it takes out any dishes that have something to do with seafood (very handy if you get sick from seafood). The expression uses the `contains()` function to check for each sequence of characters if it can be found in the value of the current element.

As you can see, the `contains()` function takes two arguments: the string to be checked and the string to be checked for. The following rules are applied to determine the result:

- The result is `true` if the first string contains the exact same sequence of Unicode characters as the second string.

- If the second string is empty, the result is always `true`.
- If the first string is empty, the result is `false` unless the second is also empty.

Caution

The XSLT specification is unclear whether empty strings should be handled as specified in the preceding paragraphs. Most, if not all, processors conform to this interpretation, but it is wise to test whether your processor of choice does as well.

When Listing 11.5 is applied to Listing 11.1, the result looks like Listing 11.6.

OUTPUT **LISTING 11.6** Result from Applying Listing 11.5 to Listing 11.1

```
Work up an Appetite
Caesar Salad $6.95

Chow Time!
Linguini al Pesto $16.95
Rack of Lamb $18.95
Ribs and Wings $16.95

To Top It Off
Dame Blanche $6.95
Chocolat Mousse $5.95
Banana Split $6.95
```

ANALYSIS In Listing 11.6, the result neatly displays all dishes that do not contain seafood (at least not in the name of the dish).

Using the `starts-with()` Function

The `contains()` function is useful if you want to check whether a value contains a sequence of characters, but you don't need to know where the sequence of characters actually occurs in the given string. In many scenarios, you actually want to know if the sequence of characters occurs at a certain position within the given string, in particular if a value starts with a certain sequence of characters. The `starts-with()` function is supplied just for this purpose. This function is particularly useful to check whether something is missing from a value, such as `http://` before a URL. Listing 11.7 shows a sample XML document with links.

LISTING 11.7 Sample XML Document with Links to Web Sites

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

```
3:   <link href="http://www.w3.org">W3 Consortium</link>
4:   <link href="www.topxml.com">Top XML</link>
5:   <link href="http://www.xml.org">The XML Industry Portal</link>
6: </index>
```

ANALYSIS The link on line 4 of Listing 11.7 does not start with `http://`, so when you create a link from it in a Web page, chances are the browser will start looking for a file named www.topxml.com on the current Web site instead of going to the other Web site. Listing 11.8 outputs a list of links in HTML to get around this problem.

LISTING 11.8 Stylesheet Dealing with Missing `http://`

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <xsl:stylesheet version="1.0"
3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:   <xsl:output method="html" encoding="UTF-8" />
6:
7:   <xsl:template match="/">
8:     <html>
9:     <body>
10:      <xsl:apply-templates />
11:    </body>
12:    </html>
13:  </xsl:template>
14:
15:  <xsl:template match="link">
16:    <a>
17:      <xsl:attribute name="href">
18:        <xsl:if test="not(starts-with(@href, 'http://'))">
19:          <xsl:text>http://</xsl:text>
20:        </xsl:if>
21:        <xsl:value-of select="@href" />
22:      </xsl:attribute>
23:      <xsl:value-of select="." />
24:    </a><br />
25:  </xsl:template>
26: </xsl:stylesheet>
```

ANALYSIS Listing 11.8 basically does nothing shocking. It just creates an HTML document with a list of links. The important stuff happens on lines 18–20, which check whether the value of the `href` attribute of the `link` element starts with `http://`. If it does not, this text is added in front of the value to be inserted into the `href` attribute of the `a` element. The result is shown in Listing 11.9.

As you can see, the `starts-with()` function takes two arguments: the string to be checked and the string to check for. If the first argument starts with the the second argument, the function returns `true`. Basically, the same rules apply as for the `contains()` function, except that the string to check for must be at the start of the string checked.

OUTPUT **LISTING 11.9** Result from Applying Listing 11.8 to Listing 11.7

```
<html>
   <body>
      <a href="http://www.w3.org">W3 Consortium</a><br>
      <a href="http://www.topxml.com">Top XML</a><br>
      <a href="http://www.xml.org">The XML Industry Portal</a><br>

   </body>
</html>
```

Both `starts-with()` and `contains()` are useful functions to check values. They are, however, not meant to perform certain tasks, such as sorting. You probably could get these functions to work, but doing so would take a lot of code. XSLT offers elements for sorting, so you should use them instead.

Tip

Don't use `starts-with()` to sort values. Instead, use the sorting and numbering constructs discussed on Day 12, "Sorting and Numbering."

You don't necessarily have to check element or attribute values with the `contains()` and `starts-with()` functions. If you use them in combination with the `name()` function, you also can check whether element or attribute names conform to certain needs. This way, you can create wildcard expressions that only match (or select) nodes with certain characters in their name. Listing 11.10 shows this principle in action.

LISTING 11.10 Stylesheet Filtering Elements on Name

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:template match="/">
9:      <xsl:apply-templates select="menu/*" />
10:   </xsl:template>
11:
```

```
12:   <xsl:template match="menu/*[starts-with(name(),'ent')]">
13:     <xsl:value-of select="concat(@title,'&#xA;')" />
14:     <xsl:apply-templates />
15:     <xsl:text>&#xA;</xsl:text>
16:   </xsl:template>
17:
18:   <xsl:template match="dish">
19:       <xsl:value-of select="." /> $<xsl:value-of select="@price" />
20:       <xsl:text>&#xA;</xsl:text>
21:   </xsl:template>
22:
23:   <xsl:template match="*" />
24: </xsl:stylesheet>
```

ANALYSIS Listing 11.10 is similar to earlier listings in that it creates a list of dishes based on the menu XML document in Listing 11.1. This time, however, you get only a list of entrees. This result has to do with line 12, which has a predicate that matches only child elements of the `menu` element that start with `ent`. In Listing 11.1, only the `entrees` element applied, but if another element—say `entradas`—were in there as well, that element also would match. This search is similar to searches in databases or on a hard disk with a search string such as `ent*`, with the `*` character being a wildcard character. The result from applying Listing 11.10 to Listing 11.1 is shown in Listing 11.11.

OUTPUT **LISTING 11.11** Result from Applying Listing 11.10 to Listing 11.1

```
Chow Time!
Grilled Salmon $19.95
Seafood Pasta $17.95
Linguini al Pesto $16.95
Rack of Lamb $18.95
Ribs and Wings $16.95
```

Getting the Length of a String

The length of a string in itself is not very valuable information. It can, however, be useful when you're determining what some output should look like. When you're outputting HTML that may not be very significant because of the way HTML deals with formatting, but in scenarios in which much more control is needed over the output, this information may be very significant. For example, say you have to deal with a fixed width for a text document. In this situation, it is likely that the length of a string is used in conjunction with other functions performing operations on strings. When you create HTML pages with forms, you can also use this function to dynamically determine the size of a text box.

You can get the length of a string by using the `string-length()` function, which takes one argument: the string for which you want to get the length. This argument is optional, however. If you do not specify it, the length of the context node is returned instead.

Suppose you have to create the menu again, but now have a fixed width that does not allow dish names to be too long. You could create a stylesheet that takes into account the length of the dish name and price. Listing 11.12 does exactly this.

LISTING 11.12 Stylesheet Creating a Menu Without Long Names

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:template match="/">
9:      <xsl:apply-templates select="menu/*" />
10:   </xsl:template>
11:
12:   <xsl:template match="menu/*">
13:     <xsl:value-of select="concat(@title,'&#xA;')" />
14:     <xsl:apply-templates />
15:     <xsl:text>&#xA;</xsl:text>
16:   </xsl:template>
17:
18:   <xsl:template match="dish">
19:     <xsl:if test="string-length() + string-length(@price) &lt; 19">
20:       <xsl:value-of select="." /> $<xsl:value-of select="@price" />
21:       <xsl:text>&#xA;</xsl:text>
22:     </xsl:if>
23:   </xsl:template>
24: </xsl:stylesheet>
```

ANALYSIS Listing 11.12 again does much of what was done in earlier samples. This time, line 19 checks whether the length of the dish name and its price exceed 18 characters. Only if the length is 18 characters or fewer does the dish make it into the menu. In the test expression, the `string-length()` function is used first to get the length of the value of the context node. This is why no argument is supplied. The second time the function is used, the `price` attribute is given as an argument because you need to add the length of that value to the length of the dish name. The result is shown in Listing 11.13.

OUTPUT **LISTING 11.13** Result from Applying Listing 11.12 to Listing 11.1

```
Work up an Appetite
Crab Cakes $8.95
Jumbo Prawns $9.95
Caesar Salad $6.95

Chow Time!
Seafood Pasta $17.95
Rack of Lamb $18.95

To Top It Off
Dame Blanche $6.95
Banana Split $6.95
```

ANALYSIS Listing 11.13 is not very wide. It contains only those dishes that meet the length requirement. One appetizer and one dessert didn't make the cut, as well as three entrees.

Note

If you want to check whether a string is empty, use the `boolean()` function instead of `string-length()`.The expressions `string-length('abc') = 0` and `boolean('abc')` yield the same result. The `boolean()` function was specifically designed for these sorts of tasks, so it is likely to perform better than the former expression.

Working with Partial Strings

The functions discussed so far have provided you with information regarding a string. When you actually want to do something with a string, you need different functions. XSLT provides several functions to obtain only a part of a string, either by specifying the position of a substring or a portion of a string that occurs before or after a certain sequence of characters. Let's look at these functions with a series of examples based on the sample XML document in Listing 11.14.

LISTING 11.14 Sample XML Document with a File List

```
<?xml version="1.0" encoding="UTF-8"?>
<folder name="My Files">
  <file>adresses.mdb</file>
  <file>basket.doc</file>
  <file>house.dwg</file>
  <file>names.xml</file>
  <file>namesout.xsl</file>
</folder>
```

ANALYSIS Listing 11.14 contains a list of files in a folder, and each file has a different file extension. The object is now to create a text file listing these files and telling you what each file type is, based on the extension. Listing 11.15 shows a stylesheet that does exactly this.

LISTING 11.15 Stylesheet Creating a File List with File Types

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:output method="text" encoding="UTF-8" />
 6:   <xsl:strip-space elements="*" />
 7:
 8:   <xsl:template match="/">
 9:     <xsl:apply-templates />
10:   </xsl:template>
11:
12:   <xsl:template match="folder">
13:     <xsl:value-of select="concat('&lt;',@name,'&gt;&#xA;')" />
14:     <xsl:apply-templates />
15:     <xsl:text>&#xA;</xsl:text>
16:   </xsl:template>
17:
18:   <xsl:template match="file">
19:     <xsl:variable name="len" select="string-length()" />
20:     <xsl:variable name="ext" select="substring(., $len - 2)" />
21:     <xsl:value-of select="concat(., '&#x20;')" />
22:     <xsl:choose>
23:       <xsl:when test="$ext = 'doc'">
24:         <xsl:text>(Word document)</xsl:text>
25:       </xsl:when>
26:       <xsl:when test="$ext = 'dwg'">
27:         <xsl:text>(AutoCad drawing)</xsl:text>
28:       </xsl:when>
29:       <xsl:when test="$ext = 'xml'">
30:         <xsl:text>(XML document)</xsl:text>
31:       </xsl:when>
32:       <xsl:when test="$ext = 'xsl'">
33:         <xsl:text>(XSL stylesheet)</xsl:text>
34:       </xsl:when>
35:     </xsl:choose>
36:     <xsl:text>&#xA;</xsl:text>
37:   </xsl:template>
38: </xsl:stylesheet>
```

ANALYSIS The template on line 12 of Listing 11.15 creates a line of output for each folder in the source XML, surrounded by < and > characters—for example, `<myfolder>`.

Line 13 takes care of this task by using the `concat()` function. Of course, you could use `xsl:text` elements, but this approach is far shorter to write and possibly easier to understand. The template that starts on line 18 deals with each file and is the place where the real action is. First, a variable is created on line 19 to hold the length of the value in the context node. Strictly speaking, a variable is not necessary because you need the length only once. On line 20, a variable is created to hold the file extension. As you can see, the value of that variable is determined with the `substring()` function. The first argument passed to it is the context node; the second argument is the starting position. Any character at or after the starting position is part of the new value. Because the last position in the string is equal to its length, and the extension is always three characters long, the starting position is the string length minus two, which is exactly what the expression says. Line 21 outputs the value of the context node and puts a space after it. Then the `xsl:choose` element on line 22 makes sure that if the extension from the file is known to the stylesheet, a full file type is shown in the output, as you can see in Listing 11.16.

OUTPUT **LISTING 11.16** Result from Applying Listing 11.15 to Listing 11.14

```
<TYXSLT21>
adresses.mdb
basket.doc (Word document)
house.dwg (AutoCad drawing)
names.xml (XML document)
namesout.xsl (XSL stylesheet)
```

ANALYSIS The result in Listing 11.16 shows that no file type is known for .mdb files. The others all show the full file type.

The `substring()` function, as used in Listing 11.15, takes two arguments: the string to get a substring from and the position to start. Note that positions in a string in XSLT start at 1 and not at 0, as is the custom in languages such as C++ and Java. Each character is counted as one character, no matter how it was encoded. So, `#xA` is counted as one character, as are Unicode surrogate pairs that go beyond the usual Unicode boundary of 65,536 characters.

The `substring()` function has one more argument, which is optional: the length of the string you want to get. By using this argument, you can get a substring from the start or middle of a string rather than only at the end. For instance, `substring('namesout.xsl',6,3)` returns `'out'`.

Getting a Substring Before or After Other Characters

Two XSLT functions that are very much related are `substring-before()` and `substring-after()`. Both functions take two arguments: the string being searched and the string to search for. If an occurrence of the second argument is found in the first, `substring-before()` returns the string up to that occurrence, excluding the occurrence itself. `substring-after()` does exactly the same, but returns the string starting after the occurring characters. If the first string does not contain an occurrence of the second argument, an empty string is returned.

Caution

If an empty string is returned, this could also mean that the first string is equal to the second or that the first string starts or ends with the second string, depending on the function you used. You can use `contains()` and `string-length()` to check whether this is the case.

Listing 11.17 performs the same task as Listing 11.15, but it has been changed in several places, among others to show the use of `substring-after()`.

LISTING 11.17 Alternative Stylesheet for Listing 11.15

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <xsl:variable name="extensions">
    <ext ext="doc">Word Document</ext>
    <ext ext="dwg">AutoCad Drawing</ext>
    <ext ext="xml">XML Document</ext>
    <ext ext="xsl">XSL Stylesheet</ext>
  </xsl:variable>

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

  <xsl:template match="folder">
    <xsl:value-of select="concat('&lt;',@name,'&gt;&#xA;')" />
    <xsl:apply-templates />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="file">
    <xsl:variable name="ext" select="substring-after(.,'.')" />
    <xsl:value-of select="concat(., '&#x20;')" />
    <xsl:if test="$extensions/ext[@ext=$ext]">
      <xsl:value-of select="concat('(',$extensions/ext[@ext=$ext],')')" />
    </xsl:if>
```

```
31:     <xsl:text>&#xA;</xsl:text>
32:   </xsl:template>
33: </xsl:stylesheet>
```

ANALYSIS Listing 11.17 takes a different road to achieve the same result as Listing 11.15. First, the template matching the `file` element starting on line 25 uses `substring-after()` to get the file's extension. As you can see on line 26, the length is no longer needed, and the same result is achieved without getting the length of the string first. Because each file consists of a name and an extension separated by a period, getting the strings after the period works just fine.

A second change to Listing 11.17 is the way the file type is retrieved for the extension. Listing 11.15 used an `xsl:choose` element for this task. In Listing 11.17, a variable named `extensions` is created on line 8. This variable contains an element for each known extension. The template matching the `file` element now just checks whether the extension is known on line 28, and if the extension is known, it selects the right extension from the variable. To make the output complete, the file type is surrounded by parentheses using the `concat()` function. Both the check and output of the extension are performed using the expression `$extensions/ext[@ext=$ext]`. For the test, this expression returns `false` if at least one node matches this expression, and the output takes the first matching element. The advantage of this approach over that of Listing 11.15 is that you don't have to add `xsl:when` elements; you can just add `ext` elements, which are much easier and clearer.

Listing 11.17 is clearcut and obvious because Listing 11.14 doesn't contain any values that have more than one period—for instance, `names.out.xsl`. In those cases, you need to be careful because both `substring-before()` and `substring-after()` operate on the first period only. Therefore, `substring-before('names.out.xsl')` returns `'names'`, and `substring-after('names.out.xsl')` returns `'out.xsl'`. The latter, of course, does not work correctly with Listing 11.17.

Replacing Parts of a String

Checking strings for contents, length, and so on is all very well, but you also may want to replace sections of strings or certain characters. You could replace parts the hard way and use `string-before()` and `string-after()` to get the job done, but you can also do it the easy way: by using the versatile function `translate()`.

The `translate()` function works differently from the other functions discussed so far. This function has three arguments:

- The original string value
- A string containing the characters that need to be replaced

- A string containing characters that should be used to replace the characters given in the second argument

The characters given in the second argument are not treated as a sequence of characters to be searched for, but each character is searched for separately and replaced by the character given to replace it. This is the character that has the same position in the third argument. So, `translate('abc','ac','AC')` returns `'AbC'`. As you can see, the letters `a` and `c` have been replaced by `A` and `C`, which have the same position in the third argument as `a` and `c` have in the second argument. Because the replacement is performed on a character-by-character basis instead of on a sequence of characters, the rules surrounding the `translate()` function are very important.

The second argument holds all the characters that need to be replaced in the source value. Each character in the source value that does not occur in that set of characters is copied to the destination as is. If the character occurs at a certain position in the list of characters to be replaced, that character is replaced with a character in the same position in the third argument, which contains the replacement characters. Hence, if the earlier expression had been `translate('abc','ac','CA')`, the result would have been `'CbA'`.

What happens when the third argument is shorter than the second? In that case, no character appears in the same position as in the list of characters to be replaced, so no character is sent to the output. For example, `translate('abc','ac','A')` returns `'Ab'`. This also means that if no replacement characters are given at all, all characters that need to be replaced are omitted from the result. If the list of replacement characters is longer than the list of characters to be replaced, the additional characters are ignored. Also, if a character occurs more than once in the list of characters to be replaced, only the first occurrence is used. Subsequent occurrences, as well as their replacement characters, are ignored. So, for example, `translate('abc','aa','AC')` returns `'Abc'`.

Listing 11.18 shows how you can use the `translate()` function.

LISTING 11.18 Using `translate()`to Create Uppercase Strings

```
<xsl:template match="file">
  <xsl:variable name="ext" select="substring-after(.,'.')" />
  <xsl:variable name="file"
       select="translate(.,'abcdefghijklmnopqrstuvwxyz'
                          ,'ABCDEFGHIJKLMNOPQRSTUVWXYZ')" />
  <xsl:value-of select="concat($file, '&#x20;')" />
  <xsl:if test="$extensions/ext[@ext=$ext]">
    <xsl:value-of select="concat('(',$extensions/ext[@ext=$ext],')')" />
  </xsl:if>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>
```

ANALYSIS Listing 11.18 shows a part of Listing 11.17, but with a change so that all files are displayed in uppercase. For this purpose, a variable named `file` is created on line 3. Its value is set using the `translate` function, which shows that all lowercase characters from *a* to *z* have to be replaced by their uppercase counterparts. On line 6, the value of the `file` variable is written to the output. You can see the result in Listing 11.19.

OUTPUT **LISTING 11.19** Result from Applying Listing 11.18 to Listing 11.14

```
<TYXSLT21>
ADRESSES.MDB
BASKET.DOC (Word Document)
HOUSE.DWG (AutoCad Drawing)
NAMES.XML (XML Document)
NAMESOUT.XSL (XSL Stylesheet)
```

Note

In this sample, any characters with accents and so on have been left out of the equation. If you need a function that creates an uppercase string for every character, you have to add all these characters, too. In that case, it might be a good idea to create a called template with a parameter called `touppercase`, for instance.

Formatting Data

If you're creating certain output, you might want to format certain values in a specific way. You may, for instance, want to format a number as currency or with a specific number of decimals. Or you may want to format a date according to a certain country's conventions. Because these data types don't exist by themselves, you have to rely on functions for formatting numbers and manipulating strings to get the job done.

Formatting Numbers

If you're working with numbers, you probably want to control what the output looks like. Especially with numbers with many digits after the decimal point, you might want to restrict the number of digits actually displayed. In addition, you might want to display numbers in a format supported by a specific country or region. To format numbers, XSLT provides the function `format-number()`, which uses the pattern you provide to format a number. This pattern defines how many decimals there should be; if any groupings should be used for thousands, millions, and so on; and what happens when a number is negative.

The basic building blocks of a pattern are `0` for a mandatory digit and `#` for an optional digit. In addition, a period is used to specify the position of the decimal point, and a comma is used for grouping. Table 11.1 shows some examples of common patterns and their result.

TABLE 11.1 Decimal Formatting Patterns

Expression	*Result*
`format-number(1234.56,'#,##0.00')`	`1,234.56`
`format-number(1234.56,'#,##0.000')`	`1,234.560`
`format-number(1234.56,'#,##0.0')`	`1,234.6`
`format-number(1234.56,'###0')`	`1235`
`format-number(-1234.56,'#,##0.00')`	`-1,234.56`
`format-number(1234.56,'###0.00')`	`1234.56`
`format-number(1234.56,'###0.0#')`	`1234.56`
`format-number(1234.5,'###0.0#')`	`1234.5`
`format-number(4.56,'00.00')`	`04.56`
`format-number(1000000,'#,##0.00')`	`1,000,000.00`
`format-number(1 div 0,'#,##0.00')`	`Infinity`
`format-number('xyz','#,##0.00')`	`NaN`

As you can see in Table 11.1, the patterns are quite simple to create. Although the samples in this table are far from complete, they should give you the general idea on what patterns do and how you can create your own.

Caution

Scientific notation is not supported in XSLT 1.0. An earlier release of MSXML did support scientific notation, but this support has been changed in more current releases. Both MSXML and Xalan now report an error, whereas Saxon just creates erroneous output.

Besides providing the pattern for a number, you also can add a prefix and suffix to the pattern. This capability is useful for working with percentages, currencies, and other known formats that have a specific meaning (such as a bank balance). The characters you can use in a prefix or suffix are bound to the same rules as normal text. The characters used to define a pattern are a problem here because you should put them between single quotation marks. However, because the pattern is likely between single quotation marks itself, using them is not possible. The `xsl:decimal-format` element provides a way around this problem, as I will show you a little later. Table 11.2 shows some common examples for prefixes and suffixes.

TABLE 11.2 Numbers Formatted with a Prefix and/or Suffix

Expression	*Result*
`format-number(1234.56,'US$ #,##0.00')`	`US$ 1,234.56`
`format-number(-1234.56,'US$ #,##0.00')`	`-US$ 1,234.56`
`format-number(1234.56,'#,##0.00 Euro')`	`1,234.56 Euro`
`format-number(-1234.56,'#,##0.00 Euro')`	`-1,234.56 Euro`
`format-number(1234.56,'$#,##0.00CR')`	`$1,234.56CR`

As you can see in Table 11.2, currencies greatly benefit from the use of prefixes and suffixes. Note that you don't need to put spaces between the number and the prefix or suffix. A downside is the position of the minus symbol to denote negative amounts. Instead of appearing after a prefix, it ends up before a prefix, which might be confusing. An option here is to insert the currency symbol in a separate `xsl:text` element so that it will always appear in front of the numbers and minus symbol.

Another option is the final weapon in the arsenal of the number pattern: being able to create two different patterns for positive and negative numbers. This allows you, among other things, to explicitly put the minus sign in front of the currency symbol if you want to do so. Another good example that can benefit from this use is numbers in a balance, where a difference needs to be shown between a positive and negative balance. Some samples are shown in Table 11.3.

TABLE 11.3 Numbers Formatted Differently for Positive and Negative Numbers

Expression	*Result*
`format-number(1234.56,'US$ #,##0.00; US$ -#,##0.00')`	`US$ 1,234.56`
`format-number(-1234.56,' US$ #,##0.00; US$ -#,##0.00')`	`US$ -1,234.56`
`format-number(1234.56,'$#,##0.00 CREDIT; $#,##0.00 DEBIT;')`	`$1,234.56 CREDIT`
`format-number(1234.56,'$#,##0.00 CREDIT; $#,##0.00 DEBIT;')`	`$1,234.56 DEBIT`

Localization

So far, the `format-number()` function has been used with two arguments: the number to be formatted and the formatting pattern. Unfortunately, this means that the output is generated in a numeric format in which the decimal separator is a period and the grouping separator a comma. Some countries use a different notation, with the comma serving as decimal separator and the period as grouping separator. To control these settings, you can use the `xsl:decimal-format` element to create a named decimal format that you can pass as a third argument to the `format-number()` function.

The `xsl:decimal-format` element is a top-level element with only attributes. You can define an alternate numeric format like this:

```
<xsl:decimal-format name="EU" decimal-separator="," grouping-separator="." />
```

This line defines a number format with the decimal separator and grouping separator as used in the European number format. When you pass this format to the `format-number()` function, you need to create the pattern in this format. Table 11.4 shows how this change would affect the values in Table 11.1.

TABLE 11.4 European Decimal Formatting Patterns

Expression	*Result*
`format-number(1234.56,'#.##0,00','EU')`	`1.234,56`
`format-number(1234.56,'#.##0,000','EU')`	`1.234,560`
`format-number(1234.56,'#.##0,0','EU')`	`1.234,6`
`format-number(1234.56,'###0','EU')`	`1235`
`format-number(-1234.56,'#.##0,00','EU')`	`-1.234,56`
`format-number(1234.56,'###0,00','EU')`	`1234,56`
`format-number(4.56,'00,00','EU')`	`04,56`
`format-number(1000000,'#.##0,00','EU')`	`1.000.000,00`
`format-number(1 div 0,'#.##0,00','EU')`	`Infinity`
`format-number('xyz','#.##0,00','EU')`	`NaN`

As you can see in Table 11.4, the value you pass along to the `format-number()` function is the same, but the patterns now have commas where there were periods, and vice versa. Each time, the `EU` number format is passed along as well. The results are now in European decimal format, except for `Infinity` and `NaN`, which are still the same.

Changing Special Number Values

As shown in the preceding tables, the default output for special number values, such as `Infinity`, stays the same with different decimal formats. You can, however, use `xsl:decimal-format` to change these values as well. You can use the `infinity` (note that this is not capitalized) and `NaN` attributes to change the values. So, this line of code

```
<xsl:decimal-format name="special" infinity="&#8734;" NaN="Invalid" />
```

yields `Invalid` for values that are not a number and ∞ for infinity.

You also can change the minus sign by using the `minus-sign` attribute like this:

```
<xsl:decimal-format name="minus" minus-sign="NEGATIVE " />
```

Now, using the `format-number()` function

```
format-number(-1234.56,'#,##0.0','minus')
```

yields `NEGATIVE 1,234.6` as a value.

Changing Default Pattern Characters

As I mentioned earlier, the characters used to specify a pattern, such as `#` and `0`, are hard to get into a prefix or suffix. To get around this problem, you can change the special characters used in the pattern. You could, for instance, change the `#` character into `@` by using the following decimal format:

```
<xsl:decimal-format name="pound" digit="#" />
```

Now a format function can use the `#` character in a prefix like

```
format-number(1234.56,'#@,@@0.0','pound')
```

to get the output `#1,234.6`.

You can change the whole set of characters shown in Table 11.5.

TABLE 11.5 Attributes to Change Pattern Characters in `format-number()`

Attribute	*Description*
`digit`	Changes the character representing a digit (`#` by default).
`zero-digit`	Changes the character representing a mandatory (or leading zero) digit (`0` by default).
`percent`	Changes the character representing the percent sign (`%` by default).
`per-mille`	Changes the character representing a per mill (or per thousand) sign (`0/00`, by default).
`pattern-separator`	Separates the pattern for positive and negative numbers (`;` by default) .

Setting the Default Format

You can create one `xsl:decimal-format` element without a name. In that case, you are overriding the default number format for the stylesheet. This capability is handy if you know that the whole stylesheet has to be in European format, for instance. You change the format like this:

```
<xsl:decimal-format decimal-separator="," grouping-separator="." />
```

If you now want to have a value in U.S. format, you need to make a named decimal format and use it with the `format-number()` function. If you do not specify a named format with the `format-number()` function, you end up with European notation.

Formatting Date And Time

XSLT does not have a date and/or time data type. Also, no functions have been specifically created to work on date and time values. Basically, when you're working with date and time, you can create your own format for use in XML source documents. As long as all documents use the same date/time format you choose, you can use string or number functions to extract the date and time and display the appropriate format.

XML Schema does have a `dateTime` type, which stores date/time values. It is nothing more than a string conforming to a set of rules. A typical `dateTime` value under the XML Schema rules looks like this:

```
2001-09-27T13:20:00-05:00
```

The numbers in front of the `T` represent the date, and the numbers after the `T` represent the time. The date format is YYYY-MM-DD. Note that the year has to be four digits, and month and day have to be two digits. The `-` character is used as separator. The time format is HH:MM:SS, with each value written as two digits. After the time, notice that another time is listed, without seconds and preceded by a minus symbol. This time represents the time zone used. For example, `-5:00` means Greenwich mean time (GMT) minus five hours, which is eastern standard time, and `+01:00` indicates the European time zone. The time zone is optional, however.

By using the format used in XML Schema, you can easily write templates that output the date and/or time in the format you want. Because this format is the most widely used date/time format, you would be wise to stick to it. Products such as SQL Server 2000 and Oracle 9i create XML from the tables in a database using this format.

The stylesheet in Listing 11.20 formats the date of a date/time value in XML Schema `dateTime` format.

LISTING 11.20 Stylesheet Formatting Date

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:variable name="monthnames">
8:      <month number="1">January</month>
9:      <month number="2">February</month>
```

```
10:     <month number="3">March</month>
11:     <month number="4">April</month>
12:     <month number="5">May</month>
13:     <month number="6">June</month>
14:     <month number="7">July</month>
15:     <month number="8">August</month>
16:     <month number="9">September</month>
17:     <month number="10">October</month>
18:     <month number="11">November</month>
19:     <month number="12">December</month>
20:   </xsl:variable>
21:
22:   <xsl:template match="/">
23:     <xsl:call-template name="formatdate" />
24:   </xsl:template>
25:
26:   <xsl:template name="formatdate">
27:     <xsl:param name="datetime" select="." />
28:     <xsl:variable name="year" select="substring-before($datetime,'-')" />
29:     <xsl:variable name="month" select="number(substring($datetime,6,2))" />
30:     <xsl:variable name="day" select="substring($datetime,9,2)" />
31:     <xsl:text>Today is </xsl:text>
32:     <xsl:value-of select="$monthnames/month[@number=$month]" />
33:     <xsl:text> </xsl:text><xsl:value-of select="$day" />
34:     <xsl:text>, </xsl:text><xsl:value-of select="$year" />
35:     <xsl:text>.</xsl:text>
36:   </xsl:template>
37: </xsl:stylesheet>
```

ANALYSIS In Listing 11.20, a called template starting on line 26 tells you today's date. That date has to come from an XML source, which is shown in Listing 11.21. The template has been made generic by creating a parameter on line 27 that takes the context element if no parameter is specified by the caller (as is the case here). That parameter is then dissected into separate variables named `day`, `month`, and `year` by using some of the string functions discussed earlier. Also, note that the month is converted to a number just to be on the safe side. The month number is used to select the month name from the variable `monthnames`, which is created on line 7. This selection is performed on line 32, which contains an expression checking the current month number against the number in the variable and displaying the one that matches. The rest of the elements surrounding it generate the text to make it look nice. Applying Listing 11.20 to Listing 11.21 yields the result shown in Listing 11.22.

LISTING 11.21 XML Source with a Date

```
<?xml version="1.0" encoding="UTF-8"?>
<date>2001-09-27T13:20:00-05:00</date>
```

OUTPUT **LISTING 11.22** Result from Applying Listing 11.20 to Listing 11.21

```
Today is September 27, 2001.
```

Formatting Other Data

By now, the general idea should be clear to you: There are no data types other than strings and numbers, so you have to manually format data with a specific meaning. This puts quite a bit of responsibility in your hands when you're creating output. If you want the output to be viewable in some way, you just format it for the output. However, when you're transforming XML for communication or data storage purposes, you also can change the format, based on what the target system expects. Because no separate data types exist, everybody can create his or her own, giving rise to incompatibility. Fortunately, XML Schema is likely to stimulate some form of uniformity, and although XSLT itself does not support XML Schema, it does benefit from this uniformity (or rather you do when writing XSLT).

Summary

Today you learned that although XSLT contains only a few data types, you can use them to create values that contain data corresponding to a data type. Using the functions that are available in XSLT to manipulate strings and numbers, you can format this data as it should be in the output. A good example is the `dateTime` type, which is defined in XML Schema but is not supported in XSLT. You can use this data type and dissect the value to show the date and time in a format that you want.

The functions `substring()`, `contains()`, and `translate()` all have an important role in these processes. In addition, `format-number()` is very important for number output for different countries and different formats for different purposes, such as accounting, computing, and so on. Other functions such as `concat()` could be omitted from XSLT because they can easily be simulated with other constructs. These functions, however, make writing expressions much easier and make way for shorter and more understandable stylesheets.

Tomorrow you will learn about sorting and numbering node-sets. You will expand upon the knowledge about number formatting you learned today to include other types of numbering, such as with letters or Roman numerals.

Q&A

Q Why isn't there a function `ends-with()`, to complement `starts-with()`?

A The people who created the XSLT (or actually XPath) specification obviously didn't think it was necessary because you can create it yourself by using `substring()` and `contains()`. Note that the same goes for `starts-with()`.

Q I want to replace sequences of characters with other sequences. The `translate()` function seems to be ill suited for this task. What do I use?

A Indeed, `translate()` isn't too handy if you want to replace sequences of characters. You need to use a combination of `translate()`, `substring()`, and `contains()` to pull off this task. This is one of the issues that XSLT 2.0 might address.

Q Being able to display different currencies is nice, but can I also do currency conversions?

A Yes, you can. You will learn more about this topic on Day 18, "Building Computational Stylesheets."

Q Are there any other date/time notations in use?

A Yes. Some types have been defined formally; others have not. The XML Schema notation is very common in XML documents. For all intents and purposes, you can view it as the standard to be used in any XML or XSLT document, even if no schema is attached to it.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is very helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: `contains()` returns a number with the position of the first occurrence of the string searched for.
2. True or False: String manipulation functions can be used only in expressions.
3. Determine the outcome of the following expression:

```
substring-after('abcxdefxgh','x')
```

4. Determine the outcome of the following expression:

```
translate('abcxdefxgh','cfx','||')
```

5. How can you create different number formats for positive and negative values?

Exercises

1. Change Listing 11.20 so that it also shows the time as `The time is 13:20 hours and 00 seconds in timezone -05:00`.
2. Create an XML file with several numbers and create a stylesheet that displays the values in different number formats. Experiment with different decimal formats.

WEEK 2

DAY 12

Sorting and Numbering

In yesterday's lesson, you learned how to manipulate string and number data values so that you can display values the way you want to. Until yesterday's lesson, you could work only with the entire element or attribute value, but with the functions discussed yesterday, you now can work with even part of a value.

Today's lesson is about sorting values in a node-set so that they are displayed in a certain order. In addition, you will learn how to add different types of numbering so that you can make nicely numbered lists. This capability is mostly useful for documents with chapters, sections, or paragraphs.

Today you will learn how to do the following:

- Sort on specific fields
- Sort in a different order
- Sort dynamically using a parameter
- Use numbering
- Create numbering in different formats

Sorting

When you select a node-set and display the results using a match template or `xsl:for-each`, the nodes are always in document order. For unstructured documents, this order is not such a problem, but when you have data that is structured, such as a set of names and addresses, sorting the data before displaying it may be very important. Fortunately, XSLT provides good support for sorting, although you need to be aware of some minor pitfalls.

Sorting basically comes in two flavors: static and dynamic. With static sorting, you know the sort order at design time, so you can create the XSLT to sort on the element or attribute you want to and in the order you want to. With dynamic sorting, you don't know the sort order at design time, but rather at runtime. This means that you need to have some way to specify the sort element or attribute and the order. Specifying the order is relatively straightforward. Specifying the element or attribute in which to sort, however, is somewhat awkward.

Using Static Sorting

You can specify a sort order by using the `xsl:sort` element, which can be used in conjunction with `xsl:for-each` or `xsl:apply-templates`. The `xsl:for-each` element is easier to use because you have a better idea of the node-set that you are actually working with; you are pulling in the node-set rather than matching nodes.

Sorting with `xsl:for-each`

If you use `xsl:sort` elements inside an `xsl:for-each` element, you need to insert these elements before any other element. You can insert an element for each value you want to sort on, with the first element being used first, then the second, and so on. This way, if the first value is the same for two nodes in a node-set, the second determines which node comes first and so on. Finally, if all values sorted on are the same, the elements are sorted in document order.

The `xsl:sort` element has several attributes. The most important one is `select`, which holds the sort key. Although this is a `select` attribute in that it can hold an expression to specify a certain sort key, this sort key needs to have bearing on the node-set that is sorted. Otherwise, it does nothing at all, and the node-set is still sorted in document order. The `select` attribute is optional, but for clarity, it is a good idea to always use this attribute. If the `select` attribute is not specified, `select="."` is implied, so it operates on the value of each node in the node-set being sorted. The other attributes are also optional and will be discussed later. Now let's look at an example that operates on Listing 12.1.

LISTING 12.1 Sample XML Document with Cars

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <car model="Camry" manufacturer="Toyota" year="1999" />
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer=" Chevrolet" year="2000" />
  <car model="Celica" manufacturer="Toyota" year="2000" />
  <car model="Mustang" manufacturer="Ford" year="2001" />
  <car model="Passat" manufacturer=" Volkswagen" year="2001" />
  <car model="Accord" manufacturer="Honda" year="2002" />
  <car model="Corvette" manufacturer="Chevrolet" year="2002" />
</cars>
```

Listing 12.1 shows a more or less familiar sight. Although you have not seen this listing like this, you're familiar with its structure. It just has some extra values. Listing 12.2 shows the stylesheet that will be used to sort Listing 12.1.

LISTING 12.2 Stylesheet Sorting Listing 12.1

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:template match="/">
8:      <xsl:for-each select="cars/car">
9:        <xsl:sort select="@manufacturer" />
10:       <xsl:sort select="@model" />
11:       <xsl:value-of select="@manufacturer" />
12:       <xsl:text> </xsl:text>
13:       <xsl:value-of select="@model" />
14:       <xsl:text> </xsl:text>
15:       <xsl:value-of select="concat('(',@year,')')" />
16:       <xsl:text>&#xA;</xsl:text>
17:     </xsl:for-each>
18:   </xsl:template>
19: </xsl:stylesheet>
```

ANALYSIS Listing 12.2 is rather straightforward. One template matches the document root. Then an `xsl:for-each` element selects and loops through all `car` elements that are child nodes of the `cars` element (that is, all of them). Next, the `xsl:sort` elements on lines 9 and 10 tell the processor that the node-set selected by the `xsl:for-each` element is to be sorted on the `manufacturer` attribute and then on the `model` attribute. All subsequent elements within the `xsl:for-each` element are then used to display the nodes.

As you can see, the `select` attribute of the `xsl:sort` element operates on the context of the nodes in the node-set. That's why `@manufacturer` does not operate on `cars` but rather on the nodes that are `car` elements.

Note

Sorting is performed *after* the node-set is constructed. This means you can sort a node-set that consists of elements scattered throughout the source document. This is also why you can sort from within an `xsl:for-each` element.

You can see the result, which is a neat list, in Listing 12.3.

OUTPUT **LISTING 12.3** Result from Applying Listing 12.2 to Listing 12.1

```
Chevrolet Corvette (2002)
Chevrolet Prizm (2000)
Ford Focus (2000)
Ford Mustang (2001)
Honda Accord (2002)
Honda Civic (2000)
Toyota Camry (1999)
Toyota Celica (2000)
Volkswagen Golf (1999)
Volkswagen Passat (2001)
```

Sorting in Descending Order

The `xsl:sort` element has an attribute named `order`, which determines the sort order of the sort key. The value of this attribute can be either `ascending` or `descending`, with `ascending` being the default, as you may have noticed from the earlier sample. Listing 12.4 shows the `xsl:sort` elements from Listing 12.2 with the sort order changed.

LISTING 12.4 Revised Stylesheet Sorting Listing 12.1

```
<xsl:sort select="@manufacturer" order="descending" />
<xsl:sort select="@model" />
```

ANALYSIS In Listing 12.4, only the first element has a sort order. This means that the `model` attribute sort key still defaults to `ascending` instead of `descending`. In other words, the second `xsl:sort` element does not take its default sort order from the element immediately above it, but rather redefines it itself. Listing 12.5 shows the output when the sort keys in Listing 12.2 are changed to those in Listing 12.4.

OUTPUT **LISTING 12.5** Result from Applying Listing 12.4 to Listing 12.1

```
Volkswagen Golf (1999)
Volkswagen Passat (2001)
Toyota Camry (1999)
Toyota Celica (2000)
Honda Accord (2002)
Honda Civic (2000)
Ford Focus (2000)
Ford Mustang (2001)
Chevrolet Corvette (2002)
Chevrolet Prizm (2000)
```

ANALYSIS In Listing 12.5, the output is now in reverse order of manufacturers. The car models, however, are still in ascending order.

NEW TERM Normally, a node-set is sorted in *document order*, which is the order in which the nodes in the node-set appear in the source document. You also can order a node-set in reverse document order by using the following sort expression:

```
<xsl:sort select="position()" order="descending" />
```

Changing Ordering Rules

So far, you've had no surprises as to how the nodes were sorted. In the preceding samples, all the values were capitalized, and hence sorted correctly. However, you also need to be aware of rules that determine what happens when lowercase and uppercase letters are mixed. For instance, does `Focus` precede `focus`, or the other way around? Unfortunately, the order depends on the default language, which in turn depends on the platform. This means that unless you specify these rules explicitly, the same stylesheet may produce different results on different computers.

Caution The XSLT specification indicates that the results of processors may differ because of their implementation and platform. So, instead of defining a common standard, XSLT allows for this inconsistency. This means that if you don't want any surprises, you have to deal with this issue in your stylesheet.

Two attributes have influence on the sorting rules: `case-order` and `lang`. The first attribute defines whether uppercase comes before lowercase, or vice versa. The second attribute defines the language settings. The former overrides the case settings of the latter, but some more rules go with the language setting. The language setting, for instance, determines whether ä is treated as a specific character to be placed after `z`, or if it is treated as a special case of `a`, in which case it just comes before `b`. Because this order depends on the language itself, it is hard to say what happens for each language (there are just too many). The best way to find out what happens in the languages you want to target is to try them on different processors. Because not even dictionaries of one language are consistent, processors are likely to be inconsistent as well.

The `case-order` attribute can have two values: `upper-first` or `lower-first`. As I said previously, the default depends on the current language, so unless you specify it, you are not sure which will be used. The following code makes sure that uppercase letters are treated first:

```
<xsl:sort select="@model" case-order="upper-first" />
```

When you use the preceding code, `Ford` comes before `ford`.

Caution

The value `upper-first` does not mean that all uppercase letters come first and then all lowercase letters, so everything is not sorted like this: `ABCabc`. It means that the uppercase version of the same letter comes before the lowercase version of that letter, so everything is sorted like this: `AaBbCc`.

The `lower-first` value, of course, means the opposite of `upper-first`. If you specify another (nonexisting) value, it is ignored and the default is used instead.

When you use the `lang` attribute, any value that yields a valid language has an impact on the sort order for specific letters. Valid values for the `lang` attribute are shown in Table 12.1.

TABLE 12.1 Valid Language Codes

Type	*Examples*
Two-letter language codes as defined in the ISO 639 standard. They may be in lowercase or uppercase characters.	`en` (English) `nl` (Dutch)
Two-letter language code, followed by a hyphen and sub-codes. The two-letter language code is the same as above. The sub-codes are as defined in the ISO 3166 standard, or those registered by Internet Assigned Numbers Authority (IANA). They are usually written in uppercase.	`en-US` (U.S. English) `nl-NL` (Netherlands Dutch)

Type	*Examples*
A language code registered with IANA, prefixed by `i-`.	`i-Klingon`
A user-defined language, preceded by `x-`.	`x-Freggle`

Table 12.1 is more than complete when it comes to sorting because it contains values that are of no use for sorting. There are other attributes, such as `xml:lang`, for which all the values in Table 12.1 are relevant. The significance of user-defined languages is dubious for sorting because the processor has no way of knowing the language settings for user-defined languages, unless they are formally described to the processor. That is obviously very unlikely.

If you want to make sure that the nodes are sorted according to U.S. English rules, your code should look as follows:

```
<xsl:sort select="@model" lang="en-US" />
```

The actual sort order depends on the implementation of the processor. The only way to find out that order is to test on a processor-by-processor basis.

Sorting on a Different Data Type

The last attribute of the `xsl:sort` element is `data-type`, which specifies the data type used to sort on. By default, this data type is `text`, but you also can use `number` if you want the value converted to a number before the ordering is done. Using this data type can be significant because 10,000 comes before 2,000 alphabetically, but not numerically. The `data-type` attribute also can contain other data types, but as yet they do nothing.

Sorting with `xsl:apply-templates`

Sorting with `xsl:apply-templates` is tricky, especially if you do not specify a `select` expression that selects nodes of only one type. The results are somewhat contrary to what you might expect. As I explained earlier, that has to do with the fact that no pull processing is involved in sorting with templates, as is the case with `xsl:for-each`. Listing 12.6 shows the code from Listing 12.1 with some modifications so that you can see the problems you can get into when sorting with `xsl:apply-templates`.

LISTING 12.6 Sample XML Source for Template Sorting

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Golf" manufacturer="Volkswagen" year="1999" />
  <model name="Camry" manufacturer="Toyota" year="1999" />
  <car model="Focus" manufacturer="Ford" year="2000" />
  <car model="Civic" manufacturer="Honda" year="2000" />
  <car model="Prizm" manufacturer="Chevrolet" year="2000" />
```

LISTING 12.6 Continued

```
  <model name="Celica" manufacturer="Toyota" year="2000" />
  <car model="Mustang" manufacturer="Ford" year="2001" />
  <model name="Passat" manufacturer="Volkswagen" year="2001" />
  <model name="Accord" manufacturer="Honda" year="2002" />
  <car model="Corvette" manufacturer="Chevrolet" year="2002" />
</cars>
```

ANALYSIS Listing 12.6 has the same cars as shown earlier but in this case has two different elements: `car` and `model`. They are in essence the same, but their names are different, and the `model` attribute in the `car` element has been replaced with the `name` attribute in the `model` element.

Now suppose you want to sort these elements like before, regardless of the actual element name. Because they have different attribute names, you might think that the stylesheet in Listing 12.7 will do the trick.

LISTING 12.7 Stylesheet Sorting with Templates

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

  <xsl:template match="cars">
    <xsl:apply-templates>
      <xsl:sort select="@manufacturer" />
      <xsl:sort select="@model" />
      <xsl:sort select="@name" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="car">
      <xsl:value-of select="@manufacturer" />
      <xsl:text> </xsl:text>
      <xsl:value-of select="@model" />
      <xsl:text> </xsl:text>
      <xsl:value-of select="concat('(',@year,')')" />
      <xsl:text>&#xA;</xsl:text>
  </xsl:template>

```

```
29:   <xsl:template match="model">
30:       <xsl:value-of select="@manufacturer" />
31:       <xsl:text> </xsl:text>
32:       <xsl:value-of select="@name" />
33:       <xsl:text>* </xsl:text>
34:       <xsl:value-of select="concat('(',@year,')')" />
35:       <xsl:text>&#xA;</xsl:text>
36:   </xsl:template>
37: </xsl:stylesheet>
```

ANALYSIS Listing 12.7 has a different template for the `car` element (line 20) and the `model` element (line 29). They do the same thing, except that for the `model` element an asterisk is added to the output so that you can see the difference. The `xsl:apply-templates` element, used on line 13, contains three `xsl:sort` elements. Judging from those elements, the cars are first ordered on manufacturer and then on either the `model` attribute or the `name` attribute, depending on which is present. Note the `xsl:strip-space` element on line 6. If it weren't present, the result would have several lines of whitespace. Listing 12.8 shows the result.

OUTPUT **LISTING 12.8** Result from Applying Listing 12.7 to Listing 12.6

```
Chevrolet Corvette (2002)
Chevrolet Prizm (2000)
Ford Focus (2000)
Ford Mustang (2001)
Honda Accord* (2002)
Honda Civic (2000)
Toyota Camry* (1999)
Toyota Celica* (2000)
Volkswagen Passat* (2001)
Volkswagen Golf (1999)
```

ANALYSIS Everything looks right in Listing 12.8, doesn't it? Well, you had better look again, and then specifically at the last two lines. Notice that these cars are in reverse order. The `model` element is used *before* the `car` element, even though the `car` element comes first alphabetically. In Listing 12.7, the `xsl:sort` element also specified the `model` attribute before the `name` attribute, so you'd think that isn't the problem either. In fact, that is the problem because the last one takes precedence in a competing scenario. So, if you have two different elements and two `xsl:sort` elements that work on only one of those elements, the last `xsl:sort` element has precedence over the former. Reversing their order would therefore yield a different result because the elements with a `model` attribute then would have precedence. Listing 12.9 shows the sorting expression from lines 13–17 of Listing 12.7 with the `name` and `model` attributes reversed.

LISTING 12.9 Sort Expression for Listing 12.7 with Reversed Attribute Order

```
<xsl:apply-templates>
  <xsl:sort select="@manufacturer" />
  <xsl:sort select="@name" />
  <xsl:sort select="@model" />
</xsl:apply-templates>
```

If you replace lines 13–17 in Listing 12.7 with the code in Listing 12.9, you get the result shown in Listing 12.10.

OUTPUT **LISTING 12.10** Result from Applying Listing 12.9 to Listing 12.6

```
Chevrolet Corvette (2002)
Chevrolet Prizm (2000)
Ford Focus (2000)
Ford Mustang (2001)
Honda Civic (2000)
Honda Accord* (2002)
Toyota Camry* (1999)
Toyota Celica* (2000)
Volkswagen Golf (1999)
Volkswagen Passat* (2001)
```

ANALYSIS You can see that reversing the attribute sort order in Listing 12.7 has an impact on the result. Although you wouldn't expect this, the result is now correct.

The problem described here is the result of a misinterpretation of the precedence rules. The question of how you can get around it remains, however, because reversing the order works in this instance, but doesn't always help. The answer is remarkably simple and lies in the select expression. Instead of using two separate elements for the different attributes, you also can make a select expression that selects both attributes. Hence, the sort condition in Listing 12.11 would yield the correct result, as shown in Listing 12.10.

LISTING 12.11 Corrected Sort Expression for Listing 12.7

```
<xsl:apply-templates>
  <xsl:sort select="@manufacturer" />
  <xsl:sort select="@model|@name" />
</xsl:apply-templates>
```

Using Dynamic Sorting

With static sorting under your belt, you can move on to dynamic sorting. Dynamic sorting isn't very hard, but the obvious way doesn't work, so you need to find another path.

The problem is that you need to define a variable or parameter to use dynamic sorting, but when you use a variable or parameter in the `select` attribute of `xsl:sort`, the result does not yield the desired effect. So, the following line does not work:

```
<xsl:sort select="$sortkey" />
```

You might think that putting the variable between curly braces will help, just as you would when you want it evaluated while you're creating dynamic attributes:

```
<xsl:sort select="{$sortkey}" />
```

Unfortunately, using curly braces is not allowed in a `select` attribute, so now what? The solution is to create an expression that checks the variable value against the name of the element or attribute you want to sort on. This solution takes some contemplation because the expression is quite tricky to produce sometimes. You must build an expression that contains the elements or attributes you need to order on and then use a predicate to filter out the specific node you need, using the `name()` function. If you want to order on an element, the expression looks like this:

```
*[name() = $sortkey]
```

This expression gets the names of all the child elements of the current context, and with the predicate, these names are compared to the value in the `sortkey` variable. You can do the same for variables by using the following expression:

```
attribute::*[name() = $sortkey]
```

This expression performs the same task as the former expression, but the `attribute` axis makes sure you compare attributes only.

The other attributes of the `xsl:sort` element fortunately can use the notation with the curly braces because these attributes do not expect a select expression, but rather a string. Listing 12.12 shows a sample using variables to determine the sort order.

LISTING 12.12 Stylesheet Using Dynamic Sorting

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:param name="sortkey1">manufacturer</xsl:param>
  <xsl:param name="sortkey2">model</xsl:param>
  <xsl:param name="order">descending</xsl:param>

  <xsl:template match="/">
    <xsl:for-each select="cars/car">
```

LISTING 12.12 Continued

```
13:       <xsl:sort select="attribute::*[name() = $sortkey1]"
14:            order="{$order}" />
15:       <xsl:sort select="attribute::*[name() = $sortkey2]" />
16:       <xsl:value-of select="@manufacturer" />
17:       <xsl:text> </xsl:text>
18:       <xsl:value-of select="@model" />
19:       <xsl:text> </xsl:text>
20:       <xsl:value-of select="concat('(',@year,')')" />
21:       <xsl:text>&#xA;</xsl:text>
22:     </xsl:for-each>
23:   </xsl:template>
24: </xsl:stylesheet>
```

ANALYSIS Listing 12.12 yields the same result as Listing 12.4, as shown in Listing 12.5. However, the result is based on the values of the sort keys and order defined as parameters on lines 7–9. So, you could change the output by adding parameters (for instance, from the command line) when you transform the source XML. Note that on lines 13 and 15 the variables `sortkey1` and `sortkey2` define the ordering in the select expression. In addition, on line 14 you set the order by putting curly braces around the parameter so that it is evaluated and its value is used. This sample uses global parameters, but you also can use local parameters, local variables, or global variables. Global parameters are specifically useful in Web sites where you want the user to be able to determine on which value should be sorted—for instance, in a shopping basket where you might want to sort on price or description.

Numbering

Numbering in XSLT is quite elaborate, and the rules surrounding it are quite complicated. I won't go into all the intricate details but will concentrate on the practical side of numbering. The following sections therefore contain many samples showing the different options you have when using numbering.

Numbering can be inserted at any place within a template or an `xsl:for-each` element but is usually used at the start of an element, particularly headings and so on. If you understand numbering using templates, you can apply that same knowledge to numbering with `xsl:for-each`, so I will not discuss this topic separately.

Inserting Numbering

You can insert numbering using the `xsl:number` element. This element has many attributes, but most are not of any interest when it comes to regular numbering. Depending on the options you use, a number is inserted at the position where you place the element. A *number* in this context is not necessarily a regular number. It also can be a Roman numeral or a letter, or it can show the section number of parent elements. So, for instance, chapter 3, section 4, paragraph 2 could get the *number* `3.4.2` or `III.iv(b)`. What the actual output looks like will be discussed in the "Controlling the Numbering Output" section later in this lesson.

The samples used here are all based on Listing 12.13, which is the familiar XML source with the menu.

LISTING 12.13 Sample XML for Numbering

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

To give you a reference, the first numbering sample, shown in Listing 12.14, is the most basic you can think of.

LISTING 12.14 Stylesheet with Basic Numbering

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

LISTING 12.14 Continued

```
10:    </xsl:template>
11:
12:   <xsl:template match="menu">
13:     <xsl:apply-templates />
14:   </xsl:template>
15:
16:   <xsl:template match="appetizers|entrees|desserts">
17:     <xsl:value-of select="concat(@title,'&#xA;')" />
18:     <xsl:apply-templates />
19:   </xsl:template>
20:
21:   <xsl:template match="dish">
22:     <xsl:text> </xsl:text>
23:     <xsl:number />
24:     <xsl:text> </xsl:text>
25:     <xsl:value-of select="." />
26:     <xsl:text>&#xA;</xsl:text>
27:   </xsl:template>
28: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 12.14 displays the menu with headers for the appetizers, entrees, and desserts. In addition, line 23 makes sure that some kind of numbering is included. Because the `xsl:number` element on line 23 has no attributes, it uses all the default values. Listing 12.15 shows the result when Listing 12.14 is applied to Listing 12.13.

OUTPUT **LISTING 12.15** Result from Applying Listing 12.14 to Listing 12.13

```
Work up an Appetite
 1 Crab Cakes
 2 Jumbo Prawns
 3 Smoked Salmon and Avocado Quesadilla
 4 Caesar Salad
Chow Time!
 1 Grilled Salmon
 2 Seafood Pasta
 3 Linguini al Pesto
 4 Rack of Lamb
 5 Ribs and Wings
To Top It Off
 1 Dame Blanche
 2 Chocolat Mousse
 3 Banana Split
```

ANALYSIS In Listing 12.15, each `dish` element is numbered according to its position related to other `dish` elements with the same parent element. So, each time a header is shown for appetizers, entrees, or desserts, the numbering starts at 1. It does so because, by default, the `level` attribute is set to single, which means that only sibling nodes are counted when the number is generated. This way of numbering is, in fact, the same as just using the `position()` function to get each node's numbers. In fact, because of the way the `xsl:number` element generates the current element's number, using the `position()` function is faster if all you have to do is simple numbering like this.

As long as you stick with templates that operate only on the context node and siblings with the same name, you are all right when it comes to numbering. Be aware, though, that when you create a template that matches more than one element, the numbering is not necessarily done on all those elements. Instead, the nodes with different names are counted separately, so you end up with numbering that is intertwined and looks erratic. Listing 12.16 shows a sample that will make this point more clear.

LISTING 12.16 Stylesheet Showing Numbering Pitfall

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:template match="/">
9:      <xsl:apply-templates />
10:    </xsl:template>
11:
12:   <xsl:template match="cars">
13:     <xsl:apply-templates />
14:   </xsl:template>
15:
16:   <xsl:template match="car|model">
17:     <xsl:number level="single" />
18:     <xsl:text> </xsl:text>
19:     <xsl:value-of select="@name|@model" />
20:     <xsl:text>&#xA;</xsl:text>
21:   </xsl:template>
22: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 12.16 is more or less the same as that in Listing 12.14, but with the exception that this stylesheet operates on Listing 12.6 and that the template on line 16 matches both `car` and `model` elements. Also, line 17 now explicitly defines `level="single"`, although this is not necessary. Line 19 outputs the value of either the `model` or `name` attribute, to work correctly for both elements. Listing 12.17 shows the result when this stylesheet is applied to Listing 12.6.

OUTPUT **LISTING 12.17** Result from Applying Listing 12.16 to Listing 12.6

```
1 Golf
1 Camry
2 Focus
3 Civic
4 Prizm
2 Celica
5 Mustang
3 Passat
4 Accord
6 Corvette
```

ANALYSIS In Listing 12.17, the numbering of the different elements is mixed. Instead of numbering through for all the elements, the elements are counted separately. Because the elements also are not sorted, the count is performed in document order, thus yielding the mixed numbering.

Fortunately, you can get around this problem by specifying which elements need to be counted. If you specify both elements, you end up with a properly numbered set. The `count` attribute serves this purpose. By default, the `count` attribute's value is the name of the context node, so the counts are separated for different elements. If you change the `xsl:number` element in Listing 12.16 into

```
<xsl:number level="single" count="car|model" />
```

the result is as shown in Listing 12.18.

OUTPUT **LISTING 12.18** Result with Count on Multiple Elements

```
1 Golf
2 Camry
3 Focus
4 Civic
5 Prizm
6 Celica
7 Mustang
8 Passat
9 Accord
10 Corvette
```

ANALYSIS In Listing 12.18, the numbering is now applied equally to both types of elements, resulting in a neatly numbered list, even though different elements are involved.

In the preceding case, you also could have used `count="*"`, but you need to be cautious with it. It might not give you the desired result. It is always a good idea to have as tight a control over numbering as is possible to avoid getting numbers that you didn't ask for.

Numbering Elements with Different Parent Elements

The examples in the preceding sections were numbered separately when elements had different parent elements. In some cases, however, you might need to number through, getting all elements that have the same name. Fortunately, the `level` attribute has more options, one of them specifically for this purpose. If you use `level="any"` in the `xsl:number` element, you number through all elements that have the same name as the context element, or the elements selected with the `select` attribute. Listing 12.19 shows what happens if you change line 23 of Listing 12.14 into the following:

```
<xsl:number level="any" />
```

OUTPUT **LISTING 12.19** Result from Number on Any Level

```
Work up an Appetite
 1 Crab Cakes
 2 Jumbo Prawns
 3 Smoked Salmon and Avocado Quesadilla
 4 Caesar Salad
Chow Time!
 5 Grilled Salmon
 6 Seafood Pasta
 7 Linguini al Pesto
 8 Rack of Lamb
 9 Ribs and Wings
To Top It Off
 10 Dame Blanche
 11 Chocolat Mousse
 12 Banana Split
```

ANALYSIS In Listing 12.19, using `level="any"` numbers from the first `dish` element to the last, regardless of each element's parent node. In fact, even if the nodes had been on different levels in the document, the numbering would still be on all nodes, sort of like using `//dish` to select all nodes in the document and then numbering them. The only difference is that in between the parent elements are also matched and handled by their template or templates.

Caution

As with `level="single"`, you need to be really careful when specifying the nodes to count in the `count` attribute. For instance, `count="*"` counts on *all* the elements in the document.

Composite Numbering

If you want to do composite numbering, like `3.4.2`, you can probably use complex expressions to get the number of parent and other ancestor elements. Fortunately, you don't have to because you can just use the `level` attribute of the `xsl:number` element and set its value to `multiple`. However, if you don't want to end up with numbering that is basically the same as using `level="single"`, you need to specify the elements you want included in the count. If you don't do that, only the siblings of the context node will be counted, and there will be no levels. The easiest way to do composite numbering is to use `count="*"`, as follows:

```
<xsl:number level="multiple" count="*" />
```

Listing 12.20 shows what happens if you change line 23 of Listing 12.14 into the preceding line.

OUTPUT **LISTING 12.20** Composite Numbering on All Elements

```
Work up an Appetite
 1.1.1 Crab Cakes
 1.1.2 Jumbo Prawns
 1.1.3 Smoked Salmon and Avocado Quesadilla
 1.1.4 Caesar Salad
Chow Time!
 1.2.1 Grilled Salmon
 1.2.2 Seafood Pasta
 1.2.3 Linguini al Pesto
 1.2.4 Rack of Lamb
 1.2.5 Ribs and Wings
To Top It Off
 1.3.1 Dame Blanche
 1.3.2 Chocolat Mousse
 1.3.3 Banana Split
```

ANALYSIS Listing 12.20 changes `level` to `multiple` and adjusts the `count` attribute. Note that because Listing 12.14 specifies numbering only for the `dish` elements, only those elements are numbered. The numbering consists of three levels, which is logical because the source XML also consists of three levels: the `menu` element, the children of the `menu` element, and the `dish` elements. As you can see, each child element of the `menu` element is numbered separately, and each time the numbering for the `dish` elements starts from the beginning.

In Listing 12.20, numbering starts at the root element, but this result is likely not what you want. After all, what's the use of having every number start with `1`? A better approach would be to start at the child elements of the root element. There are two ways to get around this problem. The first way is to change the count expression so that it omits the `menu` element (or rather numbers just the `dish` elements and their parent elements). This means changing line 23 of Listing 12.14 into the following:

```
<xsl:number level="multiple" count="dish|menu/*" />
```

This line says "Count all the `dish` elements and all the child elements of the `menu` element." The `menu` element itself is not counted, as you can see in the result in Listing 12.21.

OUTPUT **LISTING 12.21** Composite Numbering on a Subset of Elements

```
Work up an Appetite
 1.1 Crab Cakes
 1.2 Jumbo Prawns
 1.3 Smoked Salmon and Avocado Quesadilla
 1.4 Caesar Salad
Chow Time!
 2.1 Grilled Salmon
 2.2 Seafood Pasta
 2.3 Linguini al Pesto
 2.4 Rack of Lamb
 2.5 Ribs and Wings
To Top It Off
 3.1 Dame Blanche
 3.2 Chocolat Mousse
 3.3 Banana Split
```

ANALYSIS In Listing 12.21, the numbering starts at the child elements of the `menu` element. The leading `1` has completely vanished, and now each *section* is counted separately and has its own number.

You can accomplish the same result by using the `from` attribute of the `xsl:number` element. This attribute tells the processor which element or elements serve as the starting point for the composite numbering. This, too, can be an expression. To get the result in Listing 12.21, you also could change line 23 of Listing 12.14 into the following:

```
<xsl:number level="multiple" count="*" from="menu/*" />
```

Now all the elements are counted again, but the starting point is `menu/*`, which is the child element of the `menu` element. The difference between this method and the method used previously to get Listing 12.21 is that the former method gets a different result if `dish` elements are on the same level as or higher up the tree than the `menu` element. The latter method, however, ignores any elements that are above the level you're working on. This point might be very important if you're working with a document that may have the same elements in a different section, but which should be excluded. You then can use the from attribute to make sure you stick to the section you need to work on. In most cases, you should use the from attribute to change the depth of the count rather than the count attribute. That way, you can keep the count attribute simple. You should use the count attribute for this only when you can't do what you want with just the from attribute.

Note

Both the count and from attribute can be defined dynamically using a variable.

Controlling the Numbering Output

As you learned in the preceding sections, numbering in XSLT can be controlled very well and offers a good alternative to using complex expressions. In simple cases, using the position() function is often faster, but in others, xsl:number is really needed. The latter specifically applies to numbering with something other than numbers, such as Roman numerals or letters.

NEW TERM The format attribute of the xsl:number element provides numbering formats to be used when the numbering is inserted. The value of this attribute is pattern based, so you can create mixed numbering types, such as II.3.a, 2.C.1, and b.iii.i. Also, it provides the option to use something other than periods to separate the levels, so IV V I or 3.2(a) are equally possible. Basically, the format consists of two types of tokens, which are (in this context) symbols representing some function or delimiter.

The first type of token represents a numeral in some format. The second type is used for punctuation. These tokens can include periods, commas, spaces, parentheses, brackets, curly braces, and so on. By default, the formatting pattern is 1, which basically means that all numbers are represented as you have seen so far, separated by periods. All options are shown in Table 12.2.

TABLE 12.2 Number Formatting Tokens

Token	*Output*
1	1, 2, 3, and so on
	1.1, 1.2, 2.3, and so on
01	01, 02, 03, and so on
a	a, b, c, …, x, y, z, aa, ab, and so on
B	A, B, C, …, X, Y, Z, AA, AB, and so on
i	i, ii, iii, iv, …, x, xi, and so on
I	I, II, III, IV, …, X, XI, and so on

Table 12.2 is not entirely complete. Some languages contain other numerals that are represented in Unicode. For these languages, using those numbering tokens is valid. To use the numbering conventions for a certain language, you also can use the `lang` attribute, which can have the values shown earlier in Table 12.1. Another attribute that may influence numbering is `letter-value`, which can have the value `alphabetic` or `traditional`. This attribute, however, is not applicable for most languages.

Caution

> Support for language-dependent numbering is not required. It is likely that processors do not support numbering types other than those shown in Table 12.2.

The stylesheet in Listing 12.22 is based on Listing 12.14, but with some more changes, so it has composite numbering on multiple levels, with a provided format.

LISTING 12.22 Stylesheet with Formatted Composite Numbering

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

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

  <xsl:template match="appetizers|entrees|desserts">
    <xsl:number level="multiple" count="*" format="1-I(a)" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="concat(@title,'&#xA;')" />
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="dish">
    <xsl:text> </xsl:text>
    <xsl:number level="multiple" count="*" format="1-I(a)" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="." />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 12.22 inserts numbering on two levels. The first is inserted with each child element of the `menu` element, as shown on line 17 (also note the inserted whitespace on line 18). The same numbering format is used on line 25 for the `dish` elements. As you can see, both lines use a format that has three different numbering tokens and different punctuation tokens. The result is shown in Listing 12.23.

OUTPUT **LISTING 12.23** Result from Applying Listing 12.22 to Listing 12.13

```
1-I) Work up an Appetite
 1-I(a) Crab Cakes
 1-I(b) Jumbo Prawns
 1-I(c) Smoked Salmon and Avocado Quesadilla
 1-I(d) Caesar Salad
1-II) Chow Time!
 1-II(a) Grilled Salmon
 1-II(b) Seafood Pasta
 1-II(c) Linguini al Pesto
 1-II(d) Rack of Lamb
 1-II(e) Ribs and Wings
1-III) To Top It Off
 1-III(a) Dame Blanche
 1-III(b) Chocolat Mousse
 1-III(c) Banana Split
```

ANALYSIS In Listing 12.23, the different numbering tokens provide different output, numbering in the chosen format but not inserting the number or letter in the format as is. Also, you can see that the letters numbering the `dish` elements appear neatly between parentheses. Note, however, that the numbering for the child elements of the `menu` element is wrong. The opening parenthesis is missing, yet the closing parenthesis is still there. It appears this way because the format defines three numbering tokens where only two are needed. This means that before numbering, you should make sure which level you're on—for instance, by using `count(ancestor::*)` or just by knowing at what level the template will be processed.

Number Grouping

A last numbering option is grouping numbers, just as you do with large numbers when you format them with `format-number()`. The two attributes that handle this type of grouping are `grouping-size`, which is the number of characters to be grouped together, and `grouping-separator`, which is the character to be used to do the grouping. This method is similar to formatting numbers and hardly ever used, so I will not discuss it further.

Summary

Today you learned that you can sort XML elements in a node-set by using the `xsl:sort` element. You can do so statically, by defining the sort order explicitly, or dynamically, by using a variable or parameter to define the sort key and order. The latter method is somewhat tricky because of the `select` attribute. Using this method, you cannot use curly braces to get the value of a variable, so you need to use an expression instead. You can number a node-set by using `xsl:for-each` or `xsl:apply-templates`. In the latter case, you need to be cautious of side effects.

You also learned that `xsl:number` provides elaborate numbering support, which is specifically handy for documents that contain chapters, sections, and so on. You can number nodes on one level or at any level in the document. Which type you choose determines whether the numbering starts at 1 each time or numbers through the whole document. You can also create composite numbers that show the numbers of the ancestor elements.

Tomorrow you will go on a different path and learn how to split your stylesheet into separate files. This capability provides you with the opportunity to reuse partial stylesheets and use a divide-and-conquer strategy.

Q&A

Q Using `xsl:sort` with `xsl:apply-templates` caused some side effects. Do I need to be careful of any other side effects?

A No. However, it is a good thing to test sorting thoroughly before actually using it. Both sorting and numbering have many options and therefore might behave differently from what you expect, depending on the structure of the source XML.

Q Does the `xsl:number` element have any other values for the `level` attribute?

A No. At this point, only `single`, `any`, and `multiple` are valid. If you use an invalid value, the processor might report an error or default to `single`.

Q Can I use numbering on a sorted node-set?

A You can, but this approach probably won't yield the result you want. Numbering is performed based on the position of the element or elements in the document, not in the sorted node-set. You can get around this problem by sorting the node-set into a variable and then numbering on the variable, but then you lose some of the numbering options. Exercise 1 shows you what goes wrong.

Q I have seen `xsl:number` used to format a number value, not for numbering a node-set. Is that possible?

A Yes. You can use `xsl:number` to format a number. However, numbers are always converted to integers, and the options are limited. The only advantage is that you can convert a number to a character by following the same rules you use to format the numbering. In any other case, `format-number()` is the best choice.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is very helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: `xsl:sort` elements are evaluated top to bottom; the top one is always *stronger* than the bottom one.
2. True or False: Numbering with `level="single"` can apply to different elements on the same level.
3. What is the benefit of specifying a data type when sorting?
4. If you use `<xsl:sort />`, what is sorted on and in what order?
5. Does `level="multiple"` have any effect if `count="dish"`?

Exercises

1. Sort Listing 12.6 on manufacturer and model (in ascending order); then number the elements.
2. Number the menu in Listing 12.13 so that the headings are preceded by *A*, *B*, or *C* and the dishes are numbered in Roman numerals per section (no composite number).

DAY 13

Working with Multifile Stylesheets

In yesterday's lesson, the focus was on sorting the elements in a source document before sending them to the output. Lesson 12 also concentrated on all kinds of numbering, from simple numbering to composite numbering in different formats.

In today's lesson, you will learn how to create reusable stylesheets. This helps you to build stylesheets and include or import them into other stylesheets so that those stylesheets can use the functionality of the original stylesheets without having to use much code. As you will see, this enables you to build libraries of templates that perform common operations. You can then reuse these templates across many stylesheets.

Today you will learn the following:

- How to create multifile stylesheets by including or importing other stylesheets
- The difference between including and importing stylesheets
- How you can benefit from including and importing stylesheets
- The rules surrounding template precedence when you're working with included or imported templates

Using Multiple Stylesheets

Within a project or a company, you are likely to have many XML documents. Some of these documents may contain the same elements as other documents, and they may have to be handled in the same manner. Rewriting the code for these elements is probably not your idea of fun, which is why you can create stylesheets consisting of multiple files to reuse templates and other constructs.

The Benefits of Multiple Files

Obviously, working with multiple files and being able to reuse them are huge benefits, specifically when you're working in large stylesheets or with many stylesheets that need to be just slightly different for different purposes. These capabilities also allow you to work with files that have some general purpose, such as inserting a header, footer, menu, or toolbar in an HTML page. In addition, you can create a file that defines global variables to store a color scheme, font formatting, and so on for your company or a specific project. This file then can be reused across stylesheets so that each stylesheet uses the same formatting. When you need to change the formatting, changing the file that defines all the variables changes all stylesheets immediately, so all output is always consistent.

You can work with multifile stylesheets by including or importing other stylesheets. When you include a stylesheet, its contents are copied into the stylesheet including it. When you import a stylesheet, this is not the case, so you can override elements from the stylesheet you import. Depending on the elements in question, elements might be merged instead. This is of benefit, for instance, if your header color is normally blue, but in a specific case it needs to be red; the stylesheet doing the import can override the default color but still use all the other formatting definitions. This capability is very common in object-oriented programming languages such as SmallTalk, C++, and Java. If you're familiar with these languages or concepts, you are already familiar with many of the benefits.

The Drawbacks of Multiple Files

Being able to reuse already-written functionality is very powerful, but the rules governing including and importing other stylesheets are complex. These rules are complex to avoid problems from duplicated elements as much as possible. So, when you work with multiple files, you need to be very aware of all the rules involved if you want to get the output you want.

Even though these rules prevent most problems, not all includes and imports are allowed. They still cause errors. These errors might not occur at design time because errors can depend on elements in the source document. These errors are a big problem because XSLT was designed, for the most part, to never fail, as long as the syntax of the source

XML and the stylesheet are correct. Until now, all failures you encountered were design-time problems, so you could deal with them when you created the stylesheet and rest assured that your stylesheet would always work (although the result might still be wrong, of course). Now you are faced with an element that might cause runtime failures, even if your stylesheet worked properly at design time.

An additional problem is that you need to be aware of the dependencies between stylesheets. If you change a stylesheet that is imported or included by another stylesheet, the changes may cause problems in the other stylesheet. This means that if you change a stylesheet, you need to check whether the stylesheets that depend on it still work, too.

Including Stylesheets

When you include a stylesheet in another stylesheet, you are copying the contents of the `xsl:stylesheet` element of the stylesheet being included. Those contents then replace the element that is used to include the other stylesheet. The two (or more) stylesheets then basically act as though they are one stylesheet, and all elements within it have to follow the same rules as though it is just one stylesheet. This makes the order in which you include different stylesheets significant, as well as the position in the stylesheet including other stylesheets.

Note

The stylesheet that includes other stylesheets is called the *including* stylesheet, whereas the stylesheet being included is called the *included* stylesheet.

You can include one stylesheet into another stylesheet by using the `xsl:include` element. The `href` attribute, the only attribute of `xsl:include`, is mandatory and must contain a valid Uniform Resource Identifier (URI) to a document. It can be either a relative URI or an absolute URI. In most cases, you have full control over the project, so chances are you will be using only a relative URI that points either to a file within the same directory as the stylesheet including the other stylesheet or in a directory relative to that one. Only if your project involves a set of stylesheets that are distributed around several servers or if you're using stylesheets provided by a third party do you use absolute URIs. Following are some samples of the different kinds of `xsl:include` elements you might use:

```
<xsl:include href="myinclude.xsl" />

<xsl:include href="../includes/myinclude.xsl" />

<xsl:include href="http://www.somewebsite.com/xslincludes/myinclude.xsl" />
```

The first two samples are relative URIs. The first points to a file in the same directory, and the second points to a file in a sibling directory named `includes`. The last sample is an absolute URI pointing to a file on another Web site. Be aware that if the file for the stylesheet being included can't be found, the processor will raise an error.

Caution

You can include a stylesheet in a stylesheet that is included itself. The same rules apply in this situation. However, be aware that the URI is relative to the stylesheet doing the include. So, stylesheet A includes stylesheet B, which is in a different directory from stylesheet A. Then, if stylesheet B includes another stylesheet, the URI of the `xsl:include` element in that stylesheet is relative to the directory of stylesheet B, not that of stylesheet A.

The best way to get the hang of using `xsl:include` and the issues involved is to look at an example. That way, you can quickly see what's going on and how changes affect the outcome. For these samples, the familiar menu XML is again the base of operations. For quick reference, it is shown in Listing 13.1.

LISTING 13.1 Sample XML with Menu Data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

The stylesheet in Listing 13.2 creates a nice-looking menu from the XML in Listing 13.1. This stylesheet does not use `xsl:include` because you first need a stylesheet that can be included in another stylesheet.

LISTING 13.2 Stylesheet Creating an HTML Menu

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="html" encoding="utf-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:template match="appetizers|entrees|desserts">
9:      <table>
10:       <xsl:apply-templates />
11:     </table>
12:   </xsl:template>
13:
14:   <xsl:template match="dish">
15:     <tr>
16:       <td><xsl:value-of select="." /></td>
17:       <td><xsl:value-of select="concat('$',@price)" /></td>
18:     </tr>
19:   </xsl:template>
20: </xsl:stylesheet>
```

ANALYSIS Listing 13.2 creates an HTML table for each course. In that table, each `dish` element is rendered as a table row, with the name and price separated in table cells. So that this is done right, the courses are handled by one template (line 8), and a separate template on line 14 handles all the `dish` elements. Note that no template matches the root node. The stylesheet was created this way on purpose so that the template is easier to reuse and will not cause an error because two templates match the root node. Lines 5–6 make sure the output encoding and type are correct and that there is no redundant whitespace.

Caution

One of the most common mistakes with the `xsl:include` element is having two templates with the exact same match expression, such as two templates matching the root node of the source document. This situation will cause an error, so you need to create your stylesheets in such a way that this cannot occur.

Listing 13.3 shows the result from applying Listing 13.2 to Listing 13.1. Note that, by default, Saxon creates indented HTML code.

OUTPUT **LISTING 13.3** Result from Applying Listing 13.2 to Listing 13.1

```
<table>
   <tr>
      <td>Crab Cakes</td>
      <td>$8.95</td>
   </tr>
   <tr>
      <td>Jumbo Prawns</td>
      <td>$9.95</td>
   </tr>
   <tr>
      <td>Smoked Salmon and Avocado Quesadilla</td>
      <td>$10.95</td>
   </tr>
   <tr>
      <td>Caesar Salad</td>
      <td>$6.95</td>
   </tr>
</table>
<table>
   <tr>
      <td>Grilled Salmon</td>
      <td>$19.95</td>
   </tr>
   <tr>
      <td>Seafood Pasta</td>
      <td>$17.95</td>
   </tr>
   <tr>
      <td>Linguini al Pesto</td>
      <td>$16.95</td>
   </tr>
   <tr>
      <td>Rack of Lamb</td>
      <td>$18.95</td>
   </tr>
   <tr>
      <td>Ribs and Wings</td>
      <td>$16.95</td>
   </tr>
</table>
<table>
   <tr>
      <td>Dame Blanche</td>
      <td>$6.95</td>
   </tr>
   <tr>
      <td>Chocolat Mousse</td>
      <td>$5.95</td>
   </tr>
   <tr>
      <td>Banana Split</td>
      <td>$6.95</td>
   </tr>
</table>
```

ANALYSIS Listing 13.3 shows three HTML tables, one for each course. The dish name and its price are put in a separate table cell.

Listing 13.3 inserts only HTML tables; other elements such as the HTML base elements `html` and `body` are not inserted. You might conceivably have a different stylesheet that reuses the templates in Listing 13.2 and among other things adds the base HTML elements. The stylesheet in Listing 13.4 does exactly that.

LISTING 13.4 Stylesheet Including Listing 13.2

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:include href="13list02.xsl" />
6:
7:    <xsl:template match="/">
8:      <html>
9:      <body>
10:       <h1>Menu</h1>
11:       <xsl:apply-templates />
12:     </body>
13:     </html>
14:   </xsl:template>
15: </xsl:stylesheet>
```

ANALYSIS Apart from line 5 containing the `xsl:include` element, Listing 13.4 is rather straightforward. The template matching the root node inserts the base HTML code and a header. It then invokes other templates, in this case the ones that were included on line 5. Listing 13.5 shows you that the result is basically Listing 13.3 with the HTML base tags surrounding the tables and a header added before the first table starts.

OUTPUT **LISTING 13.5** Result from Applying Listing 13.4 to Listing 13.1

```
<html>
   <body>
      <h1>Menu</h1>
      <table>
         <tr>
            <td>Crab Cakes</td>
            <td>$8.95</td>
         </tr>
         <tr>
            <td>Jumbo Prawns</td>
            <td>$9.95</td>
         </tr>
         <tr>
            <td>Smoked Salmon and Avocado Quesadilla</td>
            <td>$10.95</td>
         </tr>
         <tr>
            <td>Caesar Salad</td>
            <td>$6.95</td>
         </tr>
      </table>
<!--Result truncated. The tables created are the same as Listing 13.4-->
   </body>
</html>
```

ANALYSIS Listing 13.5 has been truncated for easier reading. Basically, it shows you that the elements inserted in Listing 13.4 are added to the output of Listing 13.2 just as if Listing 13.4 contains the templates of Listing 13.2. This is, in fact, more or less the case.

As I said earlier, the top-level elements from the included stylesheet are copied over to the including stylesheet, at the position of the `xsl:include` element. So, after the `xsl:include` element in Listing 13.4 is handled by the processor, the resulting stylesheet, for all intents and purposes, looks like Listing 13.6.

Caution

The position of the `xsl:include` element in the including stylesheet is of paramount importance. A change in location will change where the elements from the included stylesheet are placed. This may have an effect on the calculated precedence of templates and other location-related issues.

LISTING 13.6 Revised Listing 13.4 with Listing 13.2 Included

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:output method="html" encoding="utf-8" />
 6:   <xsl:strip-space elements="*" />
 7:
 8:   <xsl:template match="appetizers|entrees|desserts">
 9:     <table>
10:       <xsl:apply-templates />
11:     </table>
12:   </xsl:template>
13:
14:   <xsl:template match="dish">
15:     <tr>
16:       <td><xsl:value-of select="." /></td>
17:       <td><xsl:value-of select="concat('$',@price)" /></td>
18:     </tr>
19:   </xsl:template>
20:
21:   <xsl:template match="/">
22:     <html>
23:     <body>
24:       <h1>Menu</h1>
25:       <xsl:apply-templates />
26:     </body>
27:     </html>
28:   </xsl:template>
29: </xsl:stylesheet>
```

ANALYSIS Listing 13.6 contains the top-level elements from Listing 13.2 included in Listing 13.4. Lines 5–19 are new and take the place of the original `xsl:include` element.

Because of the way the including stylesheets work, you can apply the same rules to the elements as you would to an entire stylesheet. This means that duplicate attribute-sets are merged into one, just as if you had defined two attribute-sets in the same stylesheet, as explained on Day 5, “Inserting Text and Elements.” The same goes for `xsl:output`, `xsl:strip-space`, and `xsl:preserve-space` elements, as explained on Day 7, “Controlling the Output.”

Caution

If you remember the lessons on Days 5 and 7, you also will remember that if a collision occurs between elements or attributes, the last one occurring in the resulting stylesheet wins. This point is very significant because where you include a stylesheet makes a lot of difference. Also, be aware that some processors may choose to raise an error and stop processing when certain collisions occur.

Suppose the including stylesheet defines UTF-8 as output encoding, but the included stylesheet defines UTF-16. If the `xsl:include` element occurs before the `xsl:output` element in the including stylesheet, the output is in UTF-8. However, if the `xsl:include` element comes after the `xsl:output` element, the `xsl:output` element in the included stylesheet occurs last; hence, UTF-16 is used as output encoding instead. Therefore, it is a very good idea to include other stylesheets before any other top-level elements. This way, you make sure that the elements in the including stylesheet always win if a conflict ever occurs.

Duplicate Templates

One situation that you need to be more careful with is having duplicate templates. Suppose you want to override the template dealing with the separate menu courses, so the HTML table would get a border. In that case, you could add the template in Listing 13.7 to Listing 13.4.

LISTING 13.7 Overriding Template for Courses

```
<xsl:template match="appetizers|entrees|desserts">
  <table border="1">
    <xsl:apply-templates />
  </table>
</xsl:template>
```

ANALYSIS When you add Listing 13.7 to Listing 13.6 and apply it to Listing 13.1, the processor can report an error, use one of the templates and continue, or both. If it uses the second approach, it picks the template that is last in the code. This, of course, depends on whether the `xsl:include` element is inserted before the template or after it. In the former case, the template defined in Listing 13.7 is used; in the latter case, the template in Listing 13.2 is used.

The problem here is that not all processors operate the same. MSXSL and Xalan, for instance, do not report an error, so how are you to know that something may be wrong?

Saxon, on the other hand, reports an error and continues processing, so you would spot the error. It is therefore a good idea to test with several processors before proceeding or, better still, try to avoid these situations by clever design.

Note

Day 21, "Designing XML and XSLT Applications," will deal with issues surrounding the design of stylesheets for reusability.

Duplicate Variables and Parameters

One of the more annoying problems with `xsl:include` is that duplicate variables and parameters are not allowed. This restriction again conforms to the rules explained on Day 8, "Working with Variables," and Day 9, "Working with Parameters," so it doesn't come as a real surprise. Because the same rules apply, the same scoping rules also apply, so you can still have duplicate variables and parameters in different templates because they do not share the same scope. But you cannot have duplicate global variables and parameters. If you use a central variable to store formatting information, you therefore cannot override it or merge it with another variable like you would do with attribute-sets. So, in these situations, you might want to consider using attribute-sets instead of variables.

Importing Stylesheets

Importing stylesheets into other stylesheets is another way to split them up into chunks and reuse them. You may be wondering why you would need another mechanism for this task if you can already include other stylesheets. The answer is that the rules for importing stylesheets are a lot different from those for including stylesheets. Although the basic goal is the same—reusing code—the methods differ, so you can pick the one that best suits your needs.

The Difference Between Including and Importing

When you include stylesheets, you basically create a new stylesheet. For the resulting stylesheet, the same rules apply as with any other stylesheets. Therefore, you must make sure that while you include stylesheets, you do not include any stylesheets that will make the resulting stylesheet erroneous. When you import stylesheets, however, something different happens. Instead of just copying over the code for the other stylesheets, the code is copied and given an import precedence or priority.

Note

The stylesheet that imports other stylesheets is called the *importing* stylesheet, whereas the stylesheet being imported is called the *imported* stylesheet.

For all the elements in the imported stylesheet, the precedence is lower than the elements in the importing stylesheet. This means that a template in the importing stylesheet cannot collide with a template from the imported stylesheet that matches the same node or nodes. Earlier, you learned that the same principle is not true when you include a stylesheet.

Although importing stylesheets has some advantages over including stylesheets, it is much harder to predict how the resulting stylesheet will operate. The point is that when you include stylesheets, you can just replace the `xsl:include` element with the code from the included stylesheet. Importing a stylesheet is not that simple. Predicting what the resulting stylesheet will look like and which precedence rules are in place is therefore much harder. The obvious result is that seeing what will actually happen is much harder.

Tip

Try to use `xsl:include` instead of `xsl:import` wherever it is applicable. By doing so, you can save yourself a lot of headaches from trying to determine the import precedence.

How to Import a Stylesheet

Importing a stylesheet is as easy as including one, with the sole difference that you use the `xsl:import` element. The following samples are similar to those shown for including stylesheets:

```
<xsl:import href="myimport.xsl" />

<xsl:import href="../imports/myimport.xsl" />

<xsl:import href="http://www.somewebsite.com/xslimports/myimport.xsl" />
```

As with including stylesheets, the relative paths are always calculated relative to the stylesheet doing the import, even if that stylesheet has been included or imported by another stylesheet that resides in another directory or on another server.

`xsl:import` is a top-level element, with the additional restriction that it must be inserted before any other top-level element is used. This is different from including a stylesheet, which may occur at any given point. Importing the same stylesheet more than once, either directly or indirectly, is not an error. This point is actually more significant than you might think. A stylesheet may rely on some other stylesheet when it is the root stylesheet. A new stylesheet that needs both the original stylesheets may elect to import them both, even though one of them is already being imported by the other. In doing so, you can change the import precedence of a stylesheet. This situation is depicted in Figure 13.1.

FIGURE 13.1
Import precedence with multiple imports.

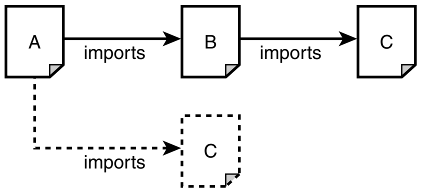
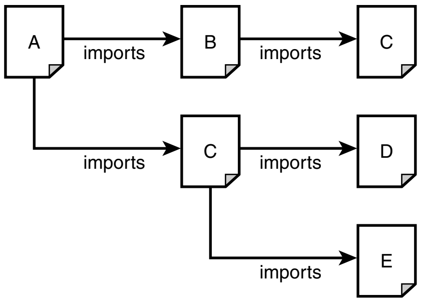


Figure 13.1 shows two situations. In the first, stylesheet A imports stylesheet B, which in turn imports stylesheet C. In this case, stylesheet C has the lowest import precedence and then stylesheet B. In the second situation, stylesheet A imports stylesheet C again after it has imported stylesheet B. Because stylesheet C is imported after stylesheet B, stylesheet C's import precedence is higher. So, when stylesheet A imports stylesheet C again, the import precedence changes.

Figure 13.2 depicts a more elaborate importing hierarchy, so you can see how import precedence really works.

FIGURE 13.2
Import precedence with hierarchy of imports.



In Figure 13.2, many imports are going on. So, how do you decide which imports have a higher precedence? The rule is that the stylesheet doing the importing has a higher precedence than the stylesheet being imported. Also, the stylesheet imported last has a higher precedence than the one imported first. The import precedence for the stylesheets in Figure 13.2 is therefore A, C, E, D, B, and again C, which comes down to A, C, E, D, and B because the extra import of C is never used.

Because of the import precedence, collisions of top-level elements are much less likely, although not impossible. So, in cases in which including a stylesheet fails, importing a stylesheet might still work. Each element is bound by its own set of import rules. The most significant are, of course, the rules surrounding templates.

Overriding Templates

For templates, the act of importing stylesheets has a lot in common with inheritance in object-oriented programming. Each stylesheet you import probably has several templates that you want to use. However, some templates may not do exactly what you want. You can *inherit* the templates that work for you and *override* the templates that don't by creating a new template with the same matching rule or name.

Overriding templates is much different from including stylesheets, where having duplicate templates can cause big problems. Earlier, Listing 13.7 was added to a stylesheet that *included* Listing 13.2. With the different processors, including the stylesheet may or may not result in an error. Listing 13.8 shows the same concept, but imports Listing 13.2 rather than includes it.

LISTING 13.8 Stylesheet Importing Listing 13.2

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:import href="13list02.xsl" />
6:
7:    <xsl:template match="/">
8:      <html>
9:      <body>
10:       <h1>Menu</h1>
11:       <xsl:apply-templates />
12:     </body>
13:     </html>
14:   </xsl:template>
15:
16:   <xsl:template match="appetizers|entrees|desserts">
17:     <table border="1">
18:       <xsl:apply-templates />
19:     </table>
20:   </xsl:template>
21: </xsl:stylesheet>
```

ANALYSIS Listing 13.8 imports Listing 13.2 on line 5. The template that creates the HTML table in Listing 13.2 is redefined in 13.8 on line 16 in Listing 13.8. Because this template has a higher import precedence, you can be sure that this template is used instead of the template in Listing 13.8. So, the resulting HTML table now has `border="1"` specified because of line 17, which is a change over Listing 13.2.

Because of the import precedence, you can import stylesheets that weren't specifically designed for code reuse. These stylesheets often contain a template matching the root node, but you can override it with your own templates. You can override all the templates that you don't want to use and use only those templates for which you wanted to import the stylesheet in the first place.

The fact that you override a template of an imported stylesheet doesn't necessarily mean that you have to re-create all the functionality of the original template. You can use the `xsl:apply-imports` element to invoke the template that matches the same node and has the next highest import precedence. This way, you can expand the original template by adding text and elements around the original result. This concept is shown in Listing 13.9.

LISTING 13.9 Stylesheet Using `xsl:apply-imports` to Invoke a Template from an Imported Stylesheet

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:import href="13list02.xsl" />
6:
7:    <xsl:template match="/">
8:      <html>
9:      <body>
10:       <h1>Menu</h1>
11:       <xsl:apply-templates />
12:     </body>
13:     </html>
14:   </xsl:template>
15:
16:   <xsl:template match="appetizers|entrees|desserts">
17:     <hr />
18:       <xsl:apply-imports />
19:     <hr />
20:   </xsl:template>
21: </xsl:stylesheet>
```

ANALYSIS Listing 13.9 is similar to Listing 13.8. It imports Listing 13.2 on line 5, and the template on line 7 inserts the base HTML elements and a header. The template on line 16 is much different, however. This template inserts a horizontal line with the HTML `hr` element at the start and end of each course. The `xsl:apply-imports` element on line 18 then invokes the template matching the same node, which is the template on line 8 of Listing 13.2. That template inserts a table for each of the courses just like before. This table is now surrounded by horizontal lines. Listing 13.10 shows the result from applying Listing 13.9 to Listing 13.1.

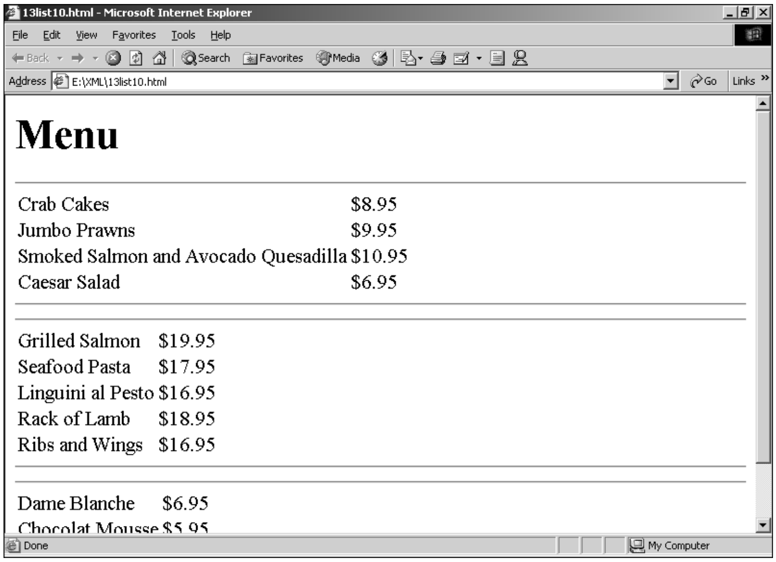
OUTPUT **LISTING 13.10** Result from Applying Listing 13.9 to Listing 13.1

```
<html>
   <body>
      <h1>Menu</h1>
      <hr>
      <table>
         <tr>
            <td>Crab Cakes</td>
            <td>$8.95</td>
         </tr>
         <tr>
            <td>Jumbo Prawns</td>
            <td>$9.95</td>
         </tr>
         <tr>
            <td>Smoked Salmon and Avocado Quesadilla</td>
            <td>$10.95</td>
         </tr>
         <tr>
            <td>Caesar Salad</td>
            <td>$6.95</td>
         </tr>
      </table>
      <hr>
      <hr>
      <table>
         <tr>
            <td>Grilled Salmon</td>
            <td>$19.95</td>
         </tr>
<!--result truncated-->
      </table>
      <hr>
<!--result truncated-->
   </body>
</html>
```

ANALYSIS Listing 13.10 is truncated for better reading, but it shows you that the result of Listing 13.9 applied to Listing 13.1 is similar to the result shown in Listing 13.5. The difference is that each table is surrounded by two horizontal lines. Figure 13.3 shows what Listing 13.10 looks like when viewed in a browser, so you can see that the horizontal lines are inserted.

FIGURE 13.3

Listing 13.10 viewed in a browser.


13list10.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\XML\13list10.html
Go Links
Done
My Computer

Menu

```
Crab Cakes                                $8.95
Jumbo Prawns                              $9.95
Smoked Salmon and Avocado Quesadilla $10.95
Caesar Salad                              $6.95

Grilled Salmon    $19.95
Seafood Pasta     $17.95
Linguini al Pesto $16.95
Rack of Lamb      $18.95
Ribs and Wings    $16.95

Dame Blanche     $6.95
Chocolat Mousse $5.95
```

If you want to change what is going on inside an imported template, using `xsl:apply-imports` doesn't help much. In that case, you still have to rewrite the whole template. The `xsl:apply-templates` element is only meant to add to the output of imported templates. The more basic the original templates are, the easier it is to alter the final output.

Tip

If you intend to use the `xsl:apply-imports` element, you should break up functionality in imported stylesheets into templates performing basic functionality. The templates that invoke the imported templates can then add more specific functionality.

Import Rules for Other Elements

Templates are the most important elements in a stylesheet because they do the actual processing. However, several other top-level elements are available, and each has its own import rules. These elements are discussed next.

Attribute-sets

When you have more than one attribute-set with the same name, the attribute-sets are merged into one. Any attribute that occurs in both attribute-sets gets the value that is defined in the attribute-set with the highest import precedence. This means that you can easily override attribute-sets from imported stylesheets because the values in the importing stylesheet's attribute-set always win.

When two attribute-sets have the same import precedence and colliding attribute values, the regular rules surrounding attribute-sets apply. This means that the processor can report an error or take the value of the last attribute-set defined. You can see how this principle works in practice by looking at Listing 13.11.

LISTING 13.11 Stylesheet Using an Attribute-set

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="html" encoding="utf-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:attribute-set name="table">
9:      <xsl:attribute name="bgcolor">#ffffcc</xsl:attribute>
10:      <xsl:attribute name="width">90%</xsl:attribute>
11:   </xsl:attribute-set>
12:
13:   <xsl:template match="appetizers|entrees|desserts">
14:     <table xsl:use-attribute-sets="table">
15:       <xsl:apply-templates />
16:     </table>
17:   </xsl:template>
18:
19:   <xsl:template match="dish">
20:     <tr>
21:       <td><xsl:value-of select="." /></td>
22:       <td><xsl:value-of select="concat('$',@price)" /></td>
23:     </tr>
24:   </xsl:template>
25: </xsl:stylesheet>
```

ANALYSIS Listing 13.11 is much like Listing 13.2, except that on line 8 an attribute-set named `table` is defined; this attribute-set is used on line 14 when creating an HTML table for each course.

You can now create another stylesheet that uses the attribute-set from Listing 13.11 and changes it somewhat. Listing 13.12 is such a stylesheet.

LISTING 13.12 Stylesheet Importing and Changing Listing 13.11

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="13list11.xsl" />

```

```
7:    <xsl:attribute-set name="table">
8:      <xsl:attribute name="bgcolor">#cccccc</xsl:attribute>
9:      <xsl:attribute name="border">1</xsl:attribute>
10:   </xsl:attribute-set>
11: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 13.12 is very small. Basically, it inherits the stylesheet in Listing 13.11, creating the exact same output, except that the attribute-set named `table` is changed on line 7. The `bgcolor` attribute on line 8 overrides the `bgcolor` attribute on line 9 of Listing 13.11. The `width` attribute is still used, and line 9 defines a new attribute named `border`. For all intents and purposes, the attribute-set that will be used when the stylesheet is processed is shown in Listing 13.13.

LISTING 13.13 Attribute-set from Merged Listings 13.11 and 13.12

```
<xsl:attribute-set name="table">
  <xsl:attribute name="bgcolor">#cccccc</xsl:attribute>
  <xsl:attribute name="width">90%</xsl:attribute>
  <xsl:attribute name="border">1</xsl:attribute>
</xsl:attribute-set>
```

ANALYSIS In Listing 13.13, the new attribute-set now has three attributes instead of two. Because the `bgcolor` attribute is overridden, the table will have a different background color than before. Your ability to merge stylesheets is useful when you have a stylesheet that already does most of what you want, but you want to change some minor details. As you can see, this process is easy, and the new stylesheet is remarkably short.

Decimal Formats

Import precedence is of no significance to the `xsl:decimal-format` element. Having more than one `xsl:decimal-format` element with the same name (unless their definitions are identical) causes an error. The processor always reports an error if this is the case, so you can handle this problem at design time.

Output Control

The `xsl:output` element behaves more or less like it would in a single stylesheet, with only one exception: If attributes that have different values are encountered, the one with the highest import precedence wins. If those elements have the same import precedence, the processor can report an error or take the last one specified. In essence, the rules for `xsl:output` follow the rules for attribute-sets.

Whitespace Handling

You handle whitespace by using the `xsl:strip-space` and `xsl:preserve-space` elements. Practically speaking, you will have conflicts if an element is matched by both of these elements. In that case, the one with the higher import precedence wins. If the import precedence rules do not give a clear winner, the matching rules explained on Day 7 are used. These rules are very much the same as template matching rules.

Because of the import precedence, you might get some surprising results: Rules that win when there is no import precedence may not win when there is. Listings 13.14 and 13.15 illustrate this point.

LISTING 13.14 Stylesheet Stripping Whitespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:strip-space elements="appetizers entrees desserts" />

  <xsl:template match="/">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 13.14 copies all the elements from the source XML but strips the whitespace in the `appetizers`, `entrees`, and `desserts` elements. Listing 13.15 imports this stylesheet.

LISTING 13.15 Stylesheet Importing Listing 13.14 and Preserving Space

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:import href="13list14.xsl" />
6:
7:    <xsl:preserve-space elements="*" />
8:
9:    <xsl:template match="/">
10:     <xsl:copy-of select="." />
11:   </xsl:template>
12: </xsl:stylesheet>
```

ANALYSIS On line 5, the stylesheet in Listing 13.15 imports Listing 13.14. As you can see, this stylesheet is imported before any other elements, as is required. Line 7 contains an `xsl:preserve-space` element telling the processor to preserve whitespace for all elements. Because this element is less specific than the `xsl:strip-space` element in Listing 13.14, the `xsl:strip-space` element would normally win over the `xsl:preserve-space` element when it comes to the elements specified in the `xsl:strip-space` element. However, because import precedence favors the `xsl:preserve-space` element, the `xsl:strip-space` element is completely ignored, and Listing 13.14 is copied over without any whitespace stripping.

In the unlikely case that a conflict occurs between elements with the same import precedence, the processor can again report an error or take the last elements specified.

Variables and Parameters

Variables and parameters are fairly straightforward. If two global variables or parameters have the same name, the one with the highest import precedence is used. Having two variables or parameters with the same name and the same import precedence causes an error, so you have to deal with this issue at design time. Because import precedence is hierarchical, this problem can occur only if you defined the same global variable (or parameter) twice or *included* a stylesheet that defines the same global variable.

Summary

In today's lesson, you learned that you can either include or import existing stylesheets. Including stylesheets is useful for breaking up functionality into reusable pieces. However, you need to make sure that no conflicts occur between elements in the included stylesheet and the including stylesheet. The basic rule you can use is that the whole stylesheet will act as one after all includes have been inserted.

Importing stylesheets is different from including stylesheets in that elements in imported stylesheets get an import precedence that defines if and when these elements are used. This way, you can take an existing, complete stylesheet and override key elements, such as templates and attribute-sets, to change the output.

Including and importing therefore have two different purposes. Including is more useful for making reusable template libraries, whereas importing is more useful when you already have stylesheets that do what you want for the most part.

Tomorrow's lesson is about the other side of the coin: using multiple source XML documents so that their data can be processed as one document. Like importing and including stylesheets can break up a stylesheet in smaller pieces that are easier to handle, being able to use multiple XML sources does the same for large XML documents.

Q&A

Q If I can import a stylesheet, I don't see why I would ever want to include one. When would I include a stylesheet rather than import it?

A When importing a stylesheet, you always have to deal with import precedence. This means that when you have a hierarchy of imports, you might inadvertently override a template you don't want to override. You can avoid this problem by including instead of importing. Including stylesheets is less complex and therefore preferred if you don't get any colliding elements.

Q How do I determine the import precedence for an element?

A Keep in mind that imports are always done first in a stylesheet and that the later an element is imported, the higher its import precedence is. The easiest way to determine import precedence is to map out the imports and includes and then determine where conflicting elements occur. Based on the map, you can determine which will win.

Q Can I see the resulting stylesheet before it is used to process a document?

A The existing processors don't allow you to see the resulting stylesheet. After tomorrow's lesson, you will be able to create a stylesheet that does this for including stylesheets. For importing stylesheets, seeing the resulting stylesheet is possible but much harder.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: A stylesheet that is included can contain a global variable with the same name as in another included stylesheet or the including stylesheet.
2. True or False: The template occurring last after all imports and includes have been done has the highest import precedence.
3. What happens when an imported stylesheet and the importing stylesheet both have an `xsl:output` element?
4. Can you include a template and then import one?
5. Does importing or including have any bearing on local variables?

Exercise

1. Create a new stylesheet that either includes or imports Listing 13.6. Make sure that it creates output from Listing 13.1 so that each course HTML table is preceded by a header containing the course title.

DAY 14

Working with Multiple XML Sources

Yesterday you learned how to create stylesheets consisting of multiple files, either by including other stylesheets or importing them. This way, you can build stylesheets from reusable pieces or alter an existing stylesheet without having to rewrite the whole thing.

Today you will learn how to use multiple XML sources, so you don't need to have all XML data in one file. Having one large XML file is often not very logical or practical, specifically when you don't need all the data that is in a file. In that case, the processor is forced to process much more data than is actually needed, which will degrade performance.

In today's lesson, you will learn how to do the following:

- Open a second XML file from a stylesheet
- Use the data from another XML source
- Dynamically define the files to be used
- Combine data from two XML sources

Accessing Other XML Sources

How do you get data from additional files? The obvious answer would be that you specify additional documents when calling the processor. This, however, is not the case; you have to specify the additional files in XSLT. You can do this by hard-coding them into the stylesheet or dynamically getting the file names from the source document. You can use the `document()` function to get data from other XML sources than the source specified when the processor was invoked.

Getting Data from an XML Source

The `document()` function can be called in different ways. In its simplest form, it takes one argument, a string naming the file to be used. This string needs to be a valid absolute or relative Uniform Resource Indicator (URI), much like the `href` attribute of the `xsl:include` and `xsl:import` elements. The following is an example of such use:

```
document('somefile.xml')
```

You can use the preceding example in any XPath expression used to select data. Listings 14.1 and 14.2 contain source XML documents that demonstrate this use.

LISTING 14.1 Sample XML Source with Cars

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <model name="Golf" manufacturer="VW" year="1999" />
  <model name="Camry" manufacturer="TY" year="1999" />
  <model name="Focus" manufacturer="FO" year="2000" />
  <model name="Civic" manufacturer="HO" year="2000" />
  <model name="Prizm" manufacturer="CV" year="2000" />
  <model name="Celica" manufacturer="TY" year="2000" />
  <model name="Mustang" manufacturer="FO" year="2001" />
  <model name="Passat" manufacturer="VW" year="2001" />
  <model name="Accord" manufacturer="HO" year="2002" />
  <model name="Corvette" manufacturer="CV" year="2002" />
</cars>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS Listing 14.1 is like most of the samples used in previous lessons. In this case, the values come from a document used on Day 8, "Working with Variables." The rest of that document has been separated and will be used later in this lesson.

LISTING 14.2 Listing with Color Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <table bgcolor="#ccccff" />
  <row bgcolor="#ffffff" />
  <altrow bgcolor="#ccccff" />
</config>
```

ANALYSIS Listing 14.2 is a configuration file holding background colors that can be used if an HTML table will be created in some stylesheet.

Listing 14.1 holds data that needs to be displayed in HTML, and Listing 14.2 holds the colors that need to be used when this is done. The stylesheet in Listing 14.3 gets those colors.

LISTING 14.3 Stylesheet Getting Colors from Listing 14.2

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <html>
    <body>
    <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="cars">
    <table bgcolor="{document('14list02.xml')/config/table/@bgcolor}">
    <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="model">
    <tr>
      <xsl:choose>
        <xsl:when test="position() mod 2 = 1">
          <xsl:attribute name="bgcolor">
            <xsl:value-of
             select="document('14list02.xml')/config/row/@bgcolor" />
          </xsl:attribute>
        </xsl:when>
```

LISTING 14.3 Continued

```
31:         <xsl:when test="position() mod 2 = 0">
32:           <xsl:attribute name="bgcolor">
33:             <xsl:value-of
34:              select="document('14list02.xml')/config/altrow/@bgcolor" />
35:           </xsl:attribute>
36:         </xsl:when>
37:       </xsl:choose>
38:       <td><xsl:value-of select="@name" /></td>
39:       <td><xsl:value-of select="year" /></td>
40:     </tr>
41:   </xsl:template>
42: </xsl:stylesheet>
```

ANALYSIS Listing 14.3 creates an HTML document with a table containing the cars in Listing 14.1. On line 16, the template matching the `cars` element creates the table and then matches the rest of the elements. On line 17, the background color for the table is pulled in from another document using the `document()` function. After the `document()` function, the precise data element is addressed with XPath just as you are used to. In fact, the only difference is that you use the `document()` function to open another XML source. In the template matching the `model` element starting on line 22, the same construction is again used on lines 28 and 34 to get the colors for rows in the table. Notice the `xsl:choose` element with `xsl:when` elements to distinguish between odd and even rows, which are shown in different colors. When you apply Listing 14.3 to Listing 14.1, you get the result shown in Listing 14.4.

OUTPUT **LISTING 14.4** Result from Applying Listing 14.3 to Listing 14.1

```
<html>
   <body>
      <table bgcolor="#ccccff">
         <tr bgcolor="#ffffff">
            <td>Golf</td>
            <td></td>
         </tr>
         <tr bgcolor="#ccccff">
            <td>Camry</td>
            <td></td>
         </tr>
         <tr bgcolor="#ffffff">
            <td>Focus</td>
            <td></td>
         </tr>
         <tr bgcolor="#ccccff">
            <td>Civic</td>
            <td></td>
         </tr>
         <tr bgcolor="#ffffff">
            <td>Prizm</td>
            <td></td>
         </tr>
         <tr bgcolor="#ccccff">
            <td>Celica</td>
            <td></td>
         </tr>
         <tr bgcolor="#ffffff">
            <td>Mustang</td>
            <td></td>
         </tr>
         <tr bgcolor="#ccccff">
            <td>Passat</td>
            <td></td>
         </tr>
         <tr bgcolor="#ffffff">
            <td>Accord</td>
            <td></td>
         </tr>
         <tr bgcolor="#ccccff">
            <td>Corvette</td>
            <td></td>
         </tr>
      </table>
   </body>
</html>
```

Analysis As you can see from the result in Listing 14.4, Listing 14.3 generates a nice HTML file with alternating colors for the table rows. These colors correspond to the ones set in Listing 14.2.

Storing Data in a Variable

In the preceding sample, the same file was called several times to get data from it. All these calls are somewhat superfluous because you can call the file once and store the entire file in a variable. You then can use that variable throughout your document, instead of using the `document()` function each time. Depending on the processor, this approach might be faster when you get a great deal of data from the additional file or files. Listing 14.5 shows how to use a variable to get the entire file.

Listing 14.5 Alternative for Listing 14.3 Using a Variable

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

```

LISTING 14.5 Continued

```
5:    <xsl:output method="html" version="4.0" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:variable name="config" select="document('14list02.xml')" />
9:
10:   <xsl:template match="/">
11:     <html>
12:     <body>
13:     <xsl:apply-templates />
14:     </body>
15:     </html>
16:   </xsl:template>
17:
18:   <xsl:template match="cars">
19:     <table bgcolor="{$config/config/table/@bgcolor}">
20:     <xsl:apply-templates />
21:     </table>
22:   </xsl:template>
23:
24:   <xsl:template match="model">
25:     <tr>
26:       <xsl:choose>
27:         <xsl:when test="position() mod 2 = 1">
28:           <xsl:attribute name="bgcolor">
29:             <xsl:value-of select="$config/config/row/@bgcolor" />
30:           </xsl:attribute>
31:         </xsl:when>
32:         <xsl:when test="position() mod 2 = 0">
33:           <xsl:attribute name="bgcolor">
34:             <xsl:value-of select="$config/config/altrow/@bgcolor" />
35:           </xsl:attribute>
36:         </xsl:when>
37:       </xsl:choose>
38:       <td><xsl:value-of select="@name" /></td>
39:       <td><xsl:value-of select="year" /></td>
40:     </tr>
41:   </xsl:template>
42: </xsl:stylesheet>
```

ANALYSIS Listing 14.5 is more or less the same as Listing 14.3; in fact, these two listings create the same output when applied to Listing 14.1. The key difference is line 8, where the XML source of Listing 14.2 is loaded into the `config` variable. Lines 19, 29, and 34, which previously called the `document()` function, now use that variable to get the data needed to show the HTML table with the right colors.

Because you can use the `document()` function in an XPath expression, you are not required to store the whole XML source that the function returns in a variable. As long as the result from the entire XPath expression is valid for a variable, you can store any subset of data from the document. You can, for example, select a node-set with nodes scattered throughout the document and make it the value of a variable. So, in the preceding example, you could have used the following expression for line 8:

```
document('14list02.xml')/config
```

The preceding expression would have resulted in a node-set consisting of the children of the `config` element. The advantage of this approach is that you need a shorter expression to get to the data because you don't need to specify the root element as well. You could shorten the expression used on line 34, for instance, to

```
$config/altrow/@bgcolor
```

This shortcut isn't very startling, but it does imply something that is much more powerful: The data in the file opened with the `document()` function does not have to be XML! As long as the processor can parse it, it can be used. Basically, the document that is read needs to be an external general parsed entity. The foremost advantage is that you can have a document with several root nodes instead of just one. Whether it is smart to create non-XML documents to get external data is debatable. It's probably a better idea to create XML documents and get only the child nodes of the root node so that you don't have to add the root node each time you address a value in the variable. In situations in which you don't have control over the format of some documents, however, you may have to read documents that are not well-formed XML.

Combining Data from Different Sources

After you have the data from a different XML source in a variable, combining it with data that is in the document being processed is straightforward. The XML document in Listing 14.6 accompanies Listing 14.1.

LISTING 14.6 Sample XML Source with Car Manufacturer Data

```
<?xml version="1.0" encoding="UTF-8"?>
<manufacturers>
  <manufacturer id="VW" name="Volkswagen" country="Germany" />
  <manufacturer id="TY" name="Toyota" country="Japan" />
  <manufacturer id="FO" name="Ford" country="USA" />
  <manufacturer id="CV" name="Chevrolet" country="USA" />
  <manufacturer id="HO" name="Honda" country="Japan" />
</manufacturers>
```

ANALYSIS Listing 14.6 holds data on manufacturers. The `id` attribute contains a unique value for each manufacturer. The `manufacturer` attribute's values in Listing 14.1 correspond to these values. Listing 14.7 combines the data in Listings 14.1 and 14.6 to show a list of cars with their manufacturers.

LISTING 14.7 Stylesheet Combining Data from Listings 14.1 and 14.6

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:variable name="manufacturers"
9:         select="document('14list06.xml')/manufacturers" />
10:
11:   <xsl:template match="/">
12:     <xsl:apply-templates />
13:   </xsl:template>
14:
15:   <xsl:template match="cars">
16:     <xsl:apply-templates />
17:   </xsl:template>
18:
19:   <xsl:template match="model">
20:     <xsl:value-of select="@name" />
21:     <xsl:text> </xsl:text>
22:     <xsl:value-of select="@year" />
23:     <xsl:text> (</xsl:text>
24:     <xsl:value-of select="$manufacturers/manufacturer[@id =
25:                           current()/@manufacturer]/@name" />
26:     <xsl:text>)&#xA;</xsl:text>
27:    </xsl:template>
28: </xsl:stylesheet>
```

ANALYSIS In Listing 14.7, the data from the manufacturers is loaded into a variable on lines 8–9. The data is loaded in such a way that the root node does not have to be used when accessing the data later. The rest of the stylesheet operates on Listing 14.1 and displays a list of values. The only exceptions are lines 24 and 25, which get the name of the manufacturer from Listing 14.6. You can't see that this data is coming from a different document. It also could be a variable containing a node-set from the document being processed. The result from applying Listing 14.7 to Listing 14.1 is shown in Listing 14.8.

OUTPUT **LISTING 14.8** Result from Applying Listing 14.7 to Listing 14.1

```
Golf 1999 (Volkswagen)
Camry 1999 (Toyota)
Focus 2000 (Ford)
Civic 2000 (Honda)
Prizm 2000 (Chevrolet)
Celica 2000 (Toyota)
Mustang 2001 (Ford)
```

```
Passat 2001 (Volkswagen)
Accord 2002 (Honda)
Corvette 2002 (Chevrolet)
```

ANALYSIS As you can see in Listing 14.8, applying Listing 14.7 results in a list of cars with manufacturer information coming from another source document.

Matching Data in a Different Source

Being able to get data from other source documents is handy, but so far you've learned only how to get data at positions where you want it. You also can use template matching to operate on data in another document. The `xsl:apply-templates` element allows matching because you can specify the data that it is supposed to use in the `select` attribute. Listing 14.9 shows how to match data this way.

LISTING 14.9 Stylesheet Matching Data in a Different Source Document

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <xsl:variable name="manufacturers" select="document('14list06.xml')" />

  <xsl:template match="/">
    <xsl:apply-templates select="$manufacturers/manufacturers" />
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:for-each select="/cars/model[@manufacturer = current()/@id]">
        <ul>
          <li>
            <xsl:value-of select="@name" />
            <xsl:text> (</xsl:text>
            <xsl:value-of select="@year" />
            <xsl:text>)</xsl:text>
          </li>
        </ul>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 14.9 loads Listing 14.6 into a variable on line 8. The template matching the root element (of Listing 14.1) then uses the `xsl:apply-templates` element not to process the rest of Listing 14.1, but to process the XML tree loaded into the `manufacturers` variable. The template on line 14 matches the `manufacturers` element in that variable, so it starts processing the element. The `xsl:for-each` element on line 15 hence processes the `manufacturer` elements in Listing 14.6. Listing 14.9 is supposed to show each car that is made by the current manufacturer. This is why line 18 selects the `model` elements from Listing 14.1 so that they can be displayed as car models for the current manufacturer. Something is wrong, however, as is shown in Listing 14.10.

OUTPUT **LISTING 14.10** Result from Applying Listing 14.9 to Listing 14.1

```
<html>
   <body>
      <h1>Auto show</h1>
      <h2>Volkswagen</h2>
      <p><i>Country: Germany</i></p>
      <h2>Toyota</h2>
      <p><i>Country: Japan</i></p>
      <h2>Ford</h2>
      <p><i>Country: USA</i></p>
      <h2>Chevrolet</h2>
      <p><i>Country: USA</i></p>
      <h2>Honda</h2>
      <p><i>Country: Japan</i></p>
   </body>
</html>
```

ANALYSIS Listing 14.10 shows the HTML that results from applying Listing 14.9 to Listing 14.1. Each manufacturer name is displayed as it should be, but what happened to the cars? Each manufacturer should have two cars listed, but it doesn't. The problem here is the context. When the template on line 14 of Listing 14.9 is invoked using the `manufacturers` variable, the current context is the *document* represented by the variable. So, the expression `/cars/model` on line 18 seeks elements in the `manufacturers` variable (Listing 14.6) that match the expression. However, there are none, as this expression was meant to seek the elements of the source document (Listing 14.1). If you want Listing 14.9 to work, you should try this procedure the other way around. That is, load Listing 14.1 into a variable and process Listing 14.6 with the stylesheet. You then can pull in the data for each car easily. This method is shown in Listing 14.11.

LISTING 14.11 Stylesheet Pulling Data from a Different Source Document

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

```
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="html" version="4.0" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:variable name="cars" select="document('14list01.xml')//cars" />
9:
10:   <xsl:template match="/">
11:     <html>
12:     <body>
13:       <h1>Auto show</h1>
14:       <xsl:apply-templates />
15:     </body>
16:     </html>
17:   </xsl:template>
18:
19:   <xsl:template match="manufacturers">
20:     <xsl:for-each select="manufacturer">
21:       <h2><xsl:value-of select="@name" /></h2>
22:       <p><i>Country: <xsl:value-of select="@country" /></i></p>
23:       <xsl:for-each select="$cars/model[@manufacturer = current()/@id]">
24:         <ul>
25:           <li>
26:             <xsl:value-of select="@name" />
27:             <xsl:text> (</xsl:text>
28:             <xsl:value-of select="@year" />
29:             <xsl:text>)</xsl:text>
30:           </li>
31:         </ul>
32:       </xsl:for-each>
33:     </xsl:for-each>
34:   </xsl:template>
35: </xsl:stylesheet>
```

ANALYSIS Listing 14.11 loads the data from the `cars` element in Listing 14.1 in the `cars` variable. Line 14 invokes matching from the root element of Listing 14.6, which invokes the template on line 19. Line 23 uses `xsl:for-each` to iterate through the `model` elements in the `cars` variable. Because the source document and the variable are both available, the `select` expression for line 23 has access to the correct car models. Listing 14.12 shows the result that was intended with the stylesheet in Listing 14.9 but that you didn't get.

OUTPUT **LISTING 14.12** Result from Applying Listing 14.11 to Listing 14.6

```
<html>
<body>
<h1>Auto show</h1>
<h2>Volkswagen</h2>
<p><i>Country: Germany</i></p>
```

LISTING 14.12 Continued

```
<ul>
<li>Golf (1999)</li>
</ul>
<ul>
<li>Passat (2001)</li>
</ul>
<h2>Toyota</h2>
<p><i>Country: Japan</i></p>
<ul>
<li>Camry (1999)</li>
</ul>
<ul>
<li>Celica (2000)</li>
</ul>
<h2>Ford</h2>
<p><i>Country: USA</i></p>
<ul>
<li>Focus (2000)</li>
</ul>
<ul>
<li>Mustang (2001)</li>
</ul>
<h2>Chevrolet</h2>
<p><i>Country: USA</i></p>
<ul>
<li>Prizm (2000)</li>
</ul>
<ul>
<li>Corvette (2002)</li>
</ul>
<h2>Honda</h2>
<p><i>Country: Japan</i></p>
<ul>
<li>Civic (2000)</li>
</ul>
<ul>
<li>Accord (2002)</li>
</ul>
</body>
</html>
```

ANALYSIS If you compare Listing 14.12 with Listing 14.10, you can see that each manufacturer now has two cars on the auto show, which previously didn't show up. They didn't show up because the data for the cars was out of context. You can see that here this is not the case.

Defining Additional Documents Dynamically

In the preceding samples, the additional documents needed to be hard-coded into the stylesheet. This situation, of course, is anything but flexible. Fortunately, you can specify the documents to be used dynamically as well. You can do so by putting references in the source document or by providing them through a global variable. The XML document in Listing 14.13 holds two file references that can be used to load documents dynamically.

LISTING 14.13 Sample XML Source with File References

```
<?xml version="1.0" encoding="UTF-8"?>
<docs>
  <file href="14list01.xml" />
  <file href="14list06.xml" />
</docs>
```

ANALYSIS Listing 14.13 holds two file references. When it is processed by a stylesheet, the stylesheet can open the specified files and process them. This procedure is shown in Listing 14.14.

LISTING 14.14 Stylesheet Loading Files in Different Variables

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

 <xsl:variable name="cars" select="document(//file[1]/@href)" />
 <xsl:variable name="manufacturers" select="document(//file[2]/@href)" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates select="$manufacturers/manufacturers" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:for-each
           select="$cars/cars/model[@manufacturer = current()/@id]">
```

LISTING 14.14 Continued

```
26:         <ul>
27:           <li>
28:             <xsl:value-of select="@name" />
29:             <xsl:text> (</xsl:text>
30:             <xsl:value-of select="@year" />
31:             <xsl:text>)</xsl:text>
32:           </li>
33:         </ul>
34:       </xsl:for-each>
35:     </xsl:for-each>
36:   </xsl:template>
37: </xsl:stylesheet>
```

ANALYSIS Listing 14.14 gets the files to be processed from the document it is applied to. These values' data is stored in separate variables on lines 8 and 9. The rest of Listing 14.14 is mostly familiar, with some minor changes to which data is actually selected. Line 15 selects the `manufacturers` element from the `manufacturer` variable for matching. When that element is matched, the template on line 20 is invoked. On line 25, this template gets data from the other variable. Listing 14.14 produces the result that was intended but went wrong with Listing 14.9. Because the data is now accessible through variables, there are no context problems.

Caution

> If you use the `document()` function to try to open a file that does not exist, an error occurs. When you use an expression, however, that expression could yield an empty value, so you open `document('')`. This is not an error, it but gives you access to the elements in the stylesheet instead. This functionality is discussed later in this lesson.

The method used in Listing 14.14 is still not optimal. Problems are bound to occur when only one file is specified in Listing 14.13. This example is, in fact, only one step better than hard-coding the filenames into the stylesheet. If you spread the data over three files, you would have a problem. Fortunately, the `document()` function is more flexible and allows you to pass a node-set with string values to it. Each value is then used to specify and open a file and get the data. All the data from each document is then stored in the one variable with which the `document()` function was used. Listing 14.15 shows the changes needed for Listing 14.12 to work with the `document()` function like this.

Listing 14.15 Specifying Files with a Node-set

```
1:  <xsl:variable name="cars" select="document(//file/@href)" />
2:
3:  <xsl:template match="/">
4:    <html>
5:    <body>
6:      <h1>Auto show</h1>
7:      <xsl:apply-templates select="$cars/manufacturers" />
8:    </body>
9:    </html>
10: </xsl:template>
```

Analysis Listing 14.15 uses only one variable on line 1. Instead of passing the `href` attribute of each `file` element to separate functions, an expression passes a node-set with only the `href` attributes of all `file` elements. In the case of Listing 14.13, this means that both files' data is stored in the `cars` variable, so `$cars/cars` selects the cars from Listing 14.1, and `$cars/manufacturers` selects the manufacturers from Listing 14.6. The only other change needed in Listing 14.14 is line 15, which you need to change so that it matches the correct variable.

Note

You also can use parameters to specify the files to be loaded through the processor. Because using parameters does not really add any significant information, I will not discuss this issue. You are, of course, free to experiment.

Linking Source Documents

If you decide to split a document into multiple documents, you might wonder how a stylesheet knows which documents should be processed together. The first step you can take is to create an element or elements in each source document that a stylesheet can use to get the related files (see Listing 14.16).

Listing 14.16 XML Source Linking to Another File with an Element

```
<?xml version="1.0" encoding="UTF-8"?>
<manufacturers>
  <file href="14list01.xml" />
  <manufacturer id="VW" name="Volkswagen" country="Germany" />
  <manufacturer id="TY" name="Toyota" country="Japan" />
  <manufacturer id="FO" name="Ford" country="USA" />
  <manufacturer id="CV" name="Chevrolet" country="USA" />
  <manufacturer id="HO" name="Honda" country="Japan" />
</manufacturers>
```

ANALYSIS Listing 14.16 is the same as Listing 14.6, except for the third line. This line refers to Listing 14.1, so a stylesheet can select the file or files that Listing 14.16 is associated with.

Listing 14.16 doesn't look nice, and it is not a good idea to clutter up your XML data with other, more or less unrelated information. This is where processing instructions can lend a helping hand. On Day 2, "Transforming Your First XML," you learned that you can use the `xml-stylesheet` processing instruction to attach a stylesheet to an XML source. With your own processing instructions, you can link XML sources so that they can be processed by a stylesheet without having to hard-code filenames in the stylesheet, as shown in Listing 14.17.

LISTING 14.17 XML Source Linking to Another File with a Processing Instruction

```
<?xml version="1.0" encoding="UTF-8"?>
<?link 14list01.xml?>
<manufacturers>
  <manufacturer id="VW" name="Volkswagen" country="Germany" />
  <manufacturer id="TY" name="Toyota" country="Japan" />
  <manufacturer id="FO" name="Ford" country="USA" />
  <manufacturer id="CV" name="Chevrolet" country="USA" />
  <manufacturer id="HO" name="Honda" country="Japan" />
</manufacturers>
```

ANALYSIS Listing 14.17 has a processing instruction on the second line. The processing instruction is named `link` but could just as well have another name that makes sense to you. The value of the processing instruction is the file you want to process with the stylesheet. If you want to specify more files, you can add more processing instructions.

NEW TERM Now you need the stylesheet to access the processing instruction instead of some element. You can do that by using a *node type test*, which is similar to a name test in a location path with an axis but selects nodes based on type rather than name. Node tests are mainly interesting to select processing instructions or comments. You can select processing instructions with the node type test `processing-instruction()`. You can provide an argument with the name of a particular processing instruction if you want to select only those instructions. In a stylesheet, you can access the file specified in Listing 14.17 by using the following expression:

```
document(/child::processing-instruction('link'))//cars
```

If you replace the expression on Line 8 of Listing 14.11 with the preceding expression, you can process Listing 14.17 and get the same result as Listing 14.11.

If you have several processing instructions linking to files, the expression `/child::processing-instruction('link')` returns a node-set. You can use that node-set to open multiple files and store them in a variable.

Note

Node type tests are not discussed further. Other node type tests are `comment()`, `node()`, and `text()`.

Specifying a Different File Location

NEW TERM As I said earlier, the `document()` function can open files in the same directory as the stylesheet, in a different directory, or on a different server altogether, depending on the URI provided. If you're using relative URIs, you also can use a second argument to specify the Base URI of the document to be loaded. The *Base URI* defines a location from which files can be opened. If a Base URI is specified as an argument for the `document()` function, the file specified in the first argument is loaded from the location specified by the Base URI.

The second argument for the `document()` function can be either a string with the Base URI or a node-set. In the latter case, the value of the first element in the node-set is used as the Base URI. In the former case, the document function could look like this:

```
document('14list01.xml','http://www.somewebsite.com/xmldata/')
```

It is not very likely that you will be using this format and even less likely that you will use the other format. Using a base URI makes sense only if all the files are on one server, which is not the server you're on. In any other case, specifying the full URI is a much better solution because it is much clearer where you're getting data from.

Accessing the Stylesheet Elements

Besides being able to get data from other XML sources, the `document()` function also allows you to access nodes in the stylesheet itself. This way, you can get to data that you store in the stylesheet rather than in a separate file, or access the stylesheet elements themselves (although that is very unlikely).

You can access the stylesheet by using the following expression:

```
document('')
```

The empty string in the preceding expression makes sure that XSLT takes the stylesheet itself as a source of data. Listing 14.18 shows this expression in action.

LISTING 14.18 Stylesheet with Internal Data

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="TYXSLT21DAYS">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <car:models>
    <car:model name="Golf" manufacturer="VW" year="1999" />
    <car:model name="Camry" manufacturer="TY" year="1999" />
    <car:model name="Focus" manufacturer="FO" year="2000" />
    <car:model name="Civic" manufacturer="HO" year="2000" />
    <car:model name="Prizm" manufacturer="CV" year="2000" />
    <car:model name="Celica" manufacturer="TY" year="2000" />
    <car:model name="Mustang" manufacturer="FO" year="2001" />
    <car:model name="Passat" manufacturer="VW" year="2001" />
    <car:model name="Accord" manufacturer="HO" year="2002" />
    <car:model name="Corvette" manufacturer="CV" year="2002" />
  </car:models>

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:for-each select="document('')/xsl:stylesheet/car:models/
➥car:model[@manufacturer = current()/@id]">
        <ul>
          <li>
            <xsl:value-of select="@name" />
            <xsl:text> (</xsl:text>
            <xsl:value-of select="@year" />
            <xsl:text>)</xsl:text>
          </li>
        </ul>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Lines 9–20 of Listing 14.8 hold data that has nothing to do with the stylesheet itself. This data is accessed on line 35 through the `document()` function. Other than that, there aren't many changes to this listing compared to earlier listings. The result is also the same.

Note

Listing 14.18 uses an extra namespace named `car`. Namespaces will be discussed in tomorrow's lesson, "Working with Namespaces."

Caution

The expression `document('')` is local to the stylesheet in which it is used. If it is used in a stylesheet that is included or imported into another, it returns only the stylesheet in which it was used, not the entire stylesheet after all includes and imports have been done.

Multidocument Pros and Cons

Working with multiple source XML documents, as well as multiple stylesheets, as discussed in yesterday's lesson, has its advantages. However, some pitfalls can make this task hard to deal with.

Multidocument Pitfalls

Addressing data in another document requires you to think carefully about the expression used to get the data, specifically if you stored the data in a variable. More often than not, you will be in a situation in which you can't quite get the data you need. In most cases, the expression is just a little off, but that minor difference might mean that you get no data at all. If you have some data, you could probably find out why you got the wrong piece of data. But if you don't get any data at all, finding out what you did wrong can take a long time.

If your expression is complex, try separating it from the other logic in your stylesheet first and making the expression as broad as possible. This way, you get more information than you need, but you can narrow it down step by step as you increase the complexity of your expression. As soon as your expression draws a blank, you know you've made a mistake, so you can back up one step. If you don't get any data at all to start with, either you are operating from the wrong context, or your expression may have an element too many or too few, possibly because you forgot about the root element or took it out when creating a variable.

Another pitfall is whitespace handling when opening another document. The loaded document should follow the same whitespace rules as the stylesheet loading it, as defined by the `xsl:strip-space` and `xsl:preserve-space` elements. However, some processors strip all whitespace-only nodes, regardless of those elements' values in the stylesheet. In most cases, stripping the whitespace is not a problem because the whitespace is not significant, but when it is, you might have a hard time getting around it. In those cases, you have to add whitespace yourself.

Multidocument Best Practices

With all the different options available to you, you might be wondering what the best way is to work with multiple stylesheets and multiple data files. You can provide additional data to stylesheets by using parameters, either single-valued or themselves containing entire XML structures; files defined at design time; or files defined at run-time. Which method or methods you use should be considered carefully. Some of these considerations belong to the discussion on Day 21, "Designing XML and XSLT Applications," but some rules are listed here:

- If there is no need to split documents, don't. Splitting logically related data into more files makes sense only in situations in which concurrent access by different users is necessary and you need to limit the amount of data that the processor needs to process.
- Formatting data is best stored in attribute-sets that can be included or imported in a stylesheet. It is not a good idea to store this data in a separate file that is not XSLT related. Attribute-sets are flexible and easy to work with. Having formatting data in an XML document is harder to work with and will probably perform worse.
- Storing data in a stylesheet is most often not a good idea. When you store data this way, the stylesheet's functionality is cluttered with data that, although relevant, should be placed in a separate data file.
- If you're splitting up stylesheets and data, split them the same way. Divide the data in separate files and create stylesheets operating on each file and structure. These separate files can be combined to perform a task. Because the stylesheets are split the same way as the data, changes to the data or the way you want to display the data are restricted to those modules dealing with that data.

Summary

Today you learned that you can use the `document()` function to open additional files with XML data, so you can use that data when processing an XML document. This capability enables you to group data in related structures, just like tables in a database. The advantage of such an approach is that you have smaller units of data and functionality to work with, and keeping the files you work with small helps performance.

The `document()` function can be used in several ways to open other files:

- One file based on a static string
- One file based on the value of some node or variables
- Multiple files based on the values in a node-set that point to different files

No matter how you open a file or files, when the data is copied to a variable, it can be accessed just like the data in the document being processed.

The `document()` function also allows access to the elements of the stylesheet itself. You can use this functionality to access data stored in the stylesheet, but doing so is not a particularly good idea. Using the `document()` function this way can, however, serve as a check to see whether some element exists and possibly what its value is. Be aware, however, that this functionality is restricted to the stylesheet in which it is used, not the resulting stylesheet from including and importing other stylesheets. You can get around this situation by using carefully named and loaded variables.

In tomorrow's lesson, you will learn more details about logical data grouping using XML namespaces. You will learn about both the benefits and drawbacks of using namespaces and how they relate to DTDs and XML Schemas.

Q&A

Q Can I use the `document()` function to import or include a stylesheet?

A No. You can import and include stylesheets only by using `xsl:import` and `xsl:import`. The `document()` function gets additional data; it therefore cannot be used to add additional templates and other XSLT elements.

Q When would I use a Base URI?

A For the most part, a Base URI is useful only in distributed scenarios in which you need to get several files from one location and in situations in which that location is variable. Otherwise, specifying the entire URI for a file is more useful.

Q Is using the `document()` function bad for performance?

A Because the processor always needs to open additional documents, using the `document()` function has some overhead in comparison with a single document with all the data. On the other hand, stylesheets may operate on different data, not needing a large portion of data. If that data needs to be processed each time and is not useful, then overhead is created there, too. The performance depends on the data structure and the amount of data.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: The `document()` function can be used only to load a variable.
2. True or False: You can combine data from different documents in one expression.
3. What happens if two documents are loaded into a variable and some of the data exists in both documents?
4. Do you need to specify a Base URI to access a file on another server?
5. If you match data in a variable, why is the data in the document being processed not available?

Exercises

1. Change Listing 14.9 so that it gives the right result.
2. Create a stylesheet that takes another stylesheet and creates a new stylesheet with all the `xsl:include` elements replaced by the top-level elements in the stylesheet to be included.

WEEK 2

In Review

This past week has been about refining your knowledge of XSLT and giving you more options and more flexibility when creating stylesheets. A subject that was key to Week 2 was being able to do the same things you did in Week 1, but with less code. Code reuse and adaptive templates are important if you don't want to write the same piece of code more than once and you have to keep re-inventing the wheel.

Overview of Bonus Project 2

This week's Bonus Project aims at combining several of the points you have learned over the last seven days. Of course, not everything covered will be used here because the Bonus Project is about a situation that may occur in a real-world application. It is unlikely that all that has been discussed will find its way into one small project.

Creating a Multifile Stylesheet with Parameters

This week's Bonus Project revolves around a wine and cheese list. Each list can be shown separately in an HTML table or together in one file. By using parameters, you can define the order key and sort order for both lists. The project consists of two separate XML documents with two stylesheets that operate on these documents separately and a third stylesheet that works on both XML documents and uses the other two stylesheets to build the two lists in one HTML file.

Starting Your Project

Your first step is to create the XML source that holds all the data about the wines. This is just your average XML source, as shown in Listing BP2.1.

LISTING BP2.1 Sample XML Source with Wine Data

```
<?xml version="1.0" encoding="UTF-8"?>
<wines>
  <wine origin="France" year="1999" color="red" type="dry">Bordeaux</wine>
  <wine origin="California" year="2000" color="red"
        type="dry">Ruby Cabernet</wine>
  <wine origin="Italy" year="2000" color="white" type="dry">Soave</wine>
  <wine origin="Italy" year="1998" color="red" type="dry">Chianti</wine>
  <wine origin="Germany" year="2000" color="white"
        type="sweet">Riesling</wine>
  <wine origin="France" year="1999" color="red" type="dry">Merlot</wine>
</wines>
```

ANALYSIS In Listing BP2.1, the name of the wine is the value of each `wine` element. Other properties that the wine may have are added as attributes. I created the source this way because `name` does not quite cover the meaning of the value of the `wine` element. `Type` may be a better choice, but it is already in use.

Building a stylesheet that creates an HTML table based on Listing BP2.1 shouldn't be too hard. To make it a little more spicy, I'm throwing some attribute-sets and parameter-based sorting into the mix (Listing BP2.2). This, of course, makes the whole thing more flexible and easier to reuse. Because writing the whole stylesheet in one go is not a good idea, however, I've broken up the process into steps. The first step yields a simple stylesheet that creates the HTML table. In the second step, the attribute-sets are added, and in the third step, the dynamic sorting is added.

LISTING BP2.2 Stylesheet Creating an HTML Table from Listing BP2.1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />

  <xsl:template match="/">
    <html>
    <body>
      <xsl:apply-templates />
    </body>
    </html>
```

```
13:   </xsl:template>
14:
15:   <xsl:template match="wines">
16:     <table>
17:       <xsl:call-template name="wineheader" />
18:       <xsl:apply-templates />
19:     </table>
20:   </xsl:template>
21:
22:   <xsl:template name="wineheader">
23:     <tr>
24:       <td><b>Name</b></td>
25:       <td><b>Origin</b></td>
26:       <td><b>Year</b></td>
27:       <td><b>Color</b></td>
28:       <td><b>Type</b></td>
29:     </tr>
30:   </xsl:template>
31:
32:   <xsl:template match="wine">
33:     <tr>
34:       <td><xsl:value-of select="." /></td>
35:       <td><xsl:value-of select="@origin" /></td>
36:       <td><xsl:value-of select="@year" /></td>
37:       <td><xsl:value-of select="@color" /></td>
38:       <td><xsl:value-of select="@type" /></td>
39:     </tr>
40:   </xsl:template>
41: </xsl:stylesheet>
```

ANALYSIS As you can see, Listing BP2.2 does nothing out of the ordinary. It contains three match templates for the root of the document, the `wines` element, and each `wine` element. On line 22, a called template creates the first line of a table, with names for each column. This template is called from line 17, before any of the `wine` elements are matched. Listing BP2.3 shows the partial output up until the first `wine` element.

OUTPUT **LISTING BP2.3** Partial Result from Applying Listing BP2.2 to Listing BP2.1

```
<html>
   <body>
      <table>
         <tr>
            <td><b>Name</b></td>
            <td><b>Origin</b></td>
            <td><b>Year</b></td>
            <td><b>Color</b></td>
            <td><b>Type</b></td>
         </tr>
```

LISTING BP2.3 Continued

```
         <tr>
            <td>Bordeaux</td>
            <td>France</td>
            <td>1999</td>
            <td>red</td>
            <td>dry</td>
         </tr>
```

ANALYSIS The result from Listing BP2.2 is just plain HTML containing a table. The first row in the table provides headers for the columns, so the words are in boldface there.

Adding Attribute-sets

Now it's time to add some attribute-sets that make the HTML look nicer. Listing BP2.4 shows these attribute-sets.

LISTING BP2.4 Attribute-sets to Be Added to Listing BP2.2

```
1:  <xsl:attribute-set name="table">
2:    <xsl:attribute name="bgcolor">#9999ff</xsl:attribute>
3:    <xsl:attribute name="width">75%</xsl:attribute>
4:  </xsl:attribute-set>
5:
6:  <xsl:attribute-set name="tableheader">
7:    <xsl:attribute name="bgcolor">#ffffcc</xsl:attribute>
8:  </xsl:attribute-set>
9:
10: <xsl:attribute-set name="tablerow">
11:   <xsl:attribute name="bgcolor">#ccccff</xsl:attribute>
12: </xsl:attribute-set>
```

ANALYSIS Listing BP2.4 defines three attribute-sets. The attribute-set on line 1 is to be used with the HTML `table` element, and it defines the background color and width of the table. For the first row in the table, a separate attribute-set is defined on line 6. It results in a different color for the first row in the table. For each table row, the attribute-set defined on line 10 is used.

To make the attribute-sets work properly, you need to add them to the elements where the attributes need to be inserted as well. So, line 16 in Listing BP2.2 needs to become

```
<table xsl:use-attribute-sets="table">
```

Line 23 has to become

```
<tr xsl:use-attribute-sets="tableheader">
```

And line 33 has to become

```
<tr xsl:use-attribute-sets="tablerow">
```

After you make all these changes, the result looks like Listing BP2.5.

OUTPUT **LISTING BP2.5** Partial Result from Applying Listing BP2.4 to Listing BP2.1

```
<html>
   <body>
      <table bgcolor="#9999ff" width="75%">
         <tr bgcolor="#ffffcc">
            <td>Name</td>
            <td>Origin</td>
            <td>Year</td>
            <td>Color</td>
            <td>Type</td>
         </tr>

         <tr bgcolor="#ccccff">
            <td>Bordeaux</td>
            <td>France</td>
            <td>1999</td>
            <td>red</td>
            <td>dry</td>
         </tr>
```

ANALYSIS There is not much difference between Listing BP2.3 and Listing BP2.5. The only differences are the attributes with the `table` element and the `tr` elements. The result looks dramatically different when viewed in a browser, however.

Adding Sorting

The final ingredient for this stylesheet is sorting. This means the `wines` template has to be changed so that it will sort the `wine` elements on the correct value and in the correct sort order. Because you need to perform a dynamic sort, where you can influence the sort key and order from outside the stylesheet, the stylesheet needs to have some global parameters. Listing BP2.6 shows all changed templates and the added parameters.

LISTING BP2.6 Changes to Listing BP2.4 for Dynamic Sorting

```
<xsl:param name="sortkey" />
<xsl:param name="order">ascending</xsl:param>

<xsl:template match="/">
  <html>
```

LISTING BP2.6 Continued

```
6:    <body>
7:      <xsl:apply-templates>
8:        <xsl:with-param name="sortkey" select="$sortkey" />
9:        <xsl:with-param name="order" select="$order" />
10:     </xsl:apply-templates>
11:   </body>
12:   </html>
13: </xsl:template>
14:
15: <xsl:template match="wines">
16:   <xsl:param name="sortkey" />
17:   <xsl:param name="order">ascending</xsl:param>
18:   <table xsl:use-attribute-sets="table">
19:     <xsl:call-template name="wineheader" />
20:     <xsl:choose>
21:       <xsl:when test="$sortkey">
22:         <xsl:apply-templates select="wine">
23:           <xsl:sort select="attribute::*[name() = $sortkey]"
24:                     order="{$order}" />
25:         </xsl:apply-templates>
26:       </xsl:when>
27:       <xsl:otherwise>
28:         <xsl:apply-templates select="wine">
29:           <xsl:sort select="." order="{$order}" />
30:         </xsl:apply-templates>
31:       </xsl:otherwise>
32:     </xsl:choose>
33:   </table>
34: </xsl:template>
```

ANALYSIS Listing BP2.6 is quite a change over the previous listings. First, two global parameters appear on lines 1 and 2. They enable the user to define the sort key and order from outside the stylesheet. Whether they are used is not significant because default values make sure that the data is sorted in default order. Second, the template matching the root node on line 4 now passes on these parameters into any matched template. You might be wondering why this is done; after all, the parameters are global. The answer is that the template matching the `wines` element needs to be self-sustaining if it is ever to be used in another stylesheet. Therefore, it uses local parameters (defined on lines 16 and 17), which need to be fed. Note that the names of the local and global parameters are equal, which is possible because of scoping rules. Besides the local parameters, the `wines` template now contains an `xsl:choose` element to deal with the difference between sorting on an attribute or the element value. If the `sortkey` parameter contains a value, it is the name of one of the attributes that needs to be used to order on. If it doesn't contain a value, the value of the `wine` element is used to sort on. In both cases, the `order` parameter is used to define the sort order. Note that in both cases the `xsl:apply-templates` element uses `select="wine"` to make sure that only wines are selected and sorted. The result of Listing BP2.6 is the same as Listing BP2.5, but the values are now sorted according to the default or passed sort key and order.

Creating the Second Stylesheet

To keep things from getting complicated, I've kept the second XML source and stylesheet much like the first set. The second XML source holds data on a selection of cheeses, as shown in Listing BP2.7.

LISTING BP2.7 Sample XML Source with Cheese Information

```
<?xml version="1.0" encoding="UTF-8"?>
<cheeses>
  <cheese name="Camembert" country="France" type="soft" />
  <cheese name="Gouda" country="Netherlands" type="semi hard" />
  <cheese name="Brie" country="France" type="soft" />
  <cheese name="Mozzarella" country="Italy" type="soft" />
  <cheese name="Feta" country="Greece" type="soft" />
  <cheese name="Port-salut" country="France" type="semi hard" />
  <cheese name="Parmigiano" country="Italy" type="hard" />
  <cheese name="Maasdam" country="Netherlands" type="hard" />
</cheeses>
```

ANALYSIS The structure of Listing BP2.7 is much like that of Listing BP2.1, apart from the element names, attribute names, and the number of attributes. Note that in Listing BP2.1 the elements also have values, whereas here all values are stored as attributes. Listing BP2.8 shows the stylesheet that operates on this XML source.

LISTING BP2.8 Stylesheet for Listing BP2.7

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />

  <xsl:attribute-set name="table">
    <xsl:attribute name="bgcolor">#ff9999</xsl:attribute>
    <xsl:attribute name="width">75%</xsl:attribute>
  </xsl:attribute-set>

  <xsl:attribute-set name="tableheader">
    <xsl:attribute name="bgcolor">#ffffcc</xsl:attribute>
  </xsl:attribute-set>

```

Listing BP2.8 Continued

```
  <xsl:attribute-set name="tablerow">
    <xsl:attribute name="bgcolor">#ffcccc</xsl:attribute>
  </xsl:attribute-set>

  <xsl:param name="sortkey">name</xsl:param>
  <xsl:param name="order">ascending</xsl:param>

  <xsl:template match="/">
    <html>
    <body>
      <xsl:apply-templates>
        <xsl:with-param name="sortkey" select="$sortkey" />
        <xsl:with-param name="order" select="$order" />
      </xsl:apply-templates>
    </body>
    </html>
  </xsl:template>

  <xsl:template match="cheeses">
    <xsl:param name="sortkey">name</xsl:param>
    <xsl:param name="order">ascending</xsl:param>
    <table xsl:use-attribute-sets="table">
      <xsl:call-template name="cheeseheader" />
      <xsl:apply-templates select="cheese">
        <xsl:sort select="attribute::*[name() = $sortkey]"
                  order="{$order}" />
      </xsl:apply-templates>
    </table>
  </xsl:template>

  <xsl:template name="cheeseheader">
    <tr xsl:use-attribute-sets="tableheader">
      <td>Name</td>
      <td>Country</td>
      <td>Type</td>
    </tr>
  </xsl:template>

  <xsl:template match="cheese">
    <tr xsl:use-attribute-sets="tablerow">
      <td><xsl:value-of select="@name" /></td>
      <td><xsl:value-of select="@country" /></td>
      <td><xsl:value-of select="@type" /></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing BP2.8 defines the same attribute-sets and parameters on lines 7 through 21 as the previous stylesheet. The only differences are the values of some (not all) of the `bgcolor` attributes. Instead of creating a blue table with a yellow header, this stylesheet creates a red table with a yellow header. In addition, the `sortkey` parameter on line 20 now has a default value, which is possible because all values are stored in attributes and not also in the element value. The template matching the `cheeses` element on line 34 corresponds to that matching the `wines` element in the previous stylesheet. Because all the values are stored in attributes, you don't need the `xsl:choose` construction used earlier. You can just use the `xsl:apply-templates` element with the values of the parameters, and that's that. Because the result is similar to the result of the earlier stylesheet, it is not shown here, but you can, of course, create the HTML file yourself.

Creating a Stylesheet for All the Data

With both separate stylesheets ready to be used, the last step in the project is to create a stylesheet that combines the two so that you can use the other stylesheets separately or in concert to create an HTML page that holds both tables. To make that page look good, the colors for both tables need to be the same. Remember that they are different with the other two stylesheets. Also, you need to have a way to define the sort key and order for both the tables. Listing BP2.9 shows what the stylesheet should look like.

LISTING BP2.9 Stylesheet Combining Two Stylesheets

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:import href="r2list08.xsl" />
6:    <xsl:import href="r2list06.xsl" />
7:
8:    <xsl:variable name="bodycolor">#cccccc</xsl:variable>
9:    <xsl:variable name="cheese" select="document('r2list07.xml')" />
10:
11:   <xsl:param name="cheesesortkey">name</xsl:param>
12:   <xsl:param name="cheeseorder">ascending</xsl:param>
13:   <xsl:param name="winesortkey" />
14:   <xsl:param name="wineorder">ascending</xsl:param>
15:
16:   <xsl:output method="html" version="4.0" encoding="UTF-8" />
17:
18:   <xsl:template match="/">
19:     <html>
20:     <body bgcolor="{$bodycolor}">
21:     <h2>Cheese</h2>
22:     <xsl:apply-templates select="$cheese/cheeses">
23:       <xsl:with-param name="sortkey" select="$cheesesortkey" />
```

LISTING BP2.9 Continued

```
24:       <xsl:with-param name="order" select="$cheeseorder" />
25:     </xsl:apply-templates>
26:     <h2>Wine</h2>
27:     <xsl:apply-templates>
28:       <xsl:with-param name="sortkey" select="$winesortkey" />
29:       <xsl:with-param name="order" select="$wineorder" />
30:     </xsl:apply-templates>
31:     </body>
32:     </html>
33:   </xsl:template>
34: </xsl:stylesheet>
```

ANALYSIS Listing BP2.9 combines the two earlier stylesheets; it starts by importing both stylesheets on lines 5 and 6. You must import the other stylesheets, because if you included the stylesheets instead, the templates matching the root element would be in conflict. On line 9, the XML data with the cheese information is loaded into a variable, so the stylesheet is supposed to work with the wine XML from Listing BP2.1. On lines 11–14, four parameters are defined: two pairs of sort key and order values, one pair for each dataset. Finally, the template on line 18 overrides the templates in the other stylesheets that match the root element. This way, both tables can be created, and the sorting parameters can be passed on to the right templates. First, the table with the cheeses is created on line 22 by matching the `cheeses` element in the `cheese` parameter. This invokes the template matching this element, which is defined in Listing BP2.8. The data in the XML source being transformed is then matched, which results in the template dealing with the `wines` element being matched.

Listing BP2.9 creates an HTML page with two tables in the same color scheme. But wait. How is that possible? The new stylesheet does not define any coloring at all. The answer is that the imported stylesheets provide the coloring scheme. The tables are colored the same because of the import precedence. The stylesheet that operates on the wine list is imported last, so the attribute-sets in that stylesheet have a higher import precedence. Because the same name is used for the attribute-sets, the tables use the same attribute-sets and thus get the same coloring.

Note

Try the different stylesheets with different parameters and view the output in a browser. This way, you can check that everything works nicely.

You can download the source code of Bonus Project 2 from the publisher's web site.

WEEK 3

At a Glance

Last week's lessons extended your knowledge of XSLT past its foundation. The knowledge of variables, parameters, data types, and techniques to work with multiple documents enables you to create real XSLT applications. This week you will learn more details about subjects needed to create XSLT applications with different types of functionality, such as stylesheets that perform calculations.

On Day 15, "Working with Namespaces," you will learn how to separate unrelated data and make sure that your stylesheet processes only the data that it needs to process. You also will learn how to create documents based on namespaces.

Day 16, "Advanced Data Selection," revisits data selection and discusses more complex types of selections. It also discusses how to get to data quickly through defined keys or ID type attributes defined in a DTD. These mechanisms enable you to select data from different parts of a document without having to use complex expressions.

On Day 17, "Using Recursion," you will learn what recursion is and how to use it. You will learn how recursion can get you around the problem that variables can't be changed while they are in scope.

Day 18, "Building Computational Stylesheets," is all about creating stylesheets that perform some kind of data processing, such as statistical data analysis. You will learn that although the capabilities of XSLT are somewhat limited, you can process data in several ways.

On Day 19, "Working with XSLT Extensions," you will learn how the XSLT language can be extended, enabling you to create stylesheets that perform functions that are very hard or impossible to achieve with regular XSLT.

Day 20, "Working with Different Processors," looks at some of the differences between the different processors and shows you how to get around these differences. It also shows you how to deal with different versions of XSLT.

Day 21, "Designing XML and XSLT Applications," finishes off this last week. It looks at XML and XSLT applications from a design point of view, showing you in which situations you should use a certain XML document structure or different XSLT structures.

After you finish Week 3, you will have a solid understanding of all that XSLT has to offer. This knowledge will enable you to create solid and flexible XSLT applications. Good luck!

DAY 15

Working with Namespaces

In yesterday's lesson, you learned to work with multiple XML sources. By loading additional documents into a variable, you can use one stylesheet to operate on separate datasets in different files. This capability is very useful if different files contain related data.

One of the problems of working with multiple data structures is that their elements may have the same name but a different meaning. In most cases, that also means that they need to be processed and displayed differently. This is where XML namespaces come in, which is the key topic of today's lesson. Throughout this book, you have informally been working with namespaces, but the time has come to bring this topic more to the foreground and formalize it.

When you process an XML document with one or more namepaces, the stylesheet needs to be aware of those namespaces and address the data in the source document differently from documents that don't use namespaces. Creating documents that use namespaces from XSLT is also different. These topics are covered today.

In today's lesson, you will learn the following:

- What XML namespaces are
- The benefits of namespaces
- The relationship between namespaces, DTDs, and XML Schemas
- How to utilize namespace information
- How to change and remove namespace information

Understanding Namespaces

NEW TERM When you're working with multiple source XML documents, elements and attributes with the same name can collide. Names are said to *collide* if the name is the same, but the meaning is different.

The problem with colliding elements and attributes is that because their meaning is different, they probably should be handled differently. However, because the names are the same, there is no way to figure out which meaning the element or attribute has. Namespaces are a mechanism to avoid naming collisions, so elements with the same name but with different meaning can be treated as different and, as such, be processed differently. This section takes a closer look at the hows and whys of namespaces.

Namespaces Explained

A namespace is a means to keep different vocabularies of elements apart. The mechanism for this is quite simple and is implemented in a very handy way. How namespaces work exactly is explained later in this section. First, let's look a little closer at XML vocabularies.

Understanding XML Vocabularies

Different XML structures use different vocabularies of elements. HTML (or rather XHTML, which is XML) has a vocabulary containing all the common elements in HTML. Any unknown element is not understood by the browser and basically is ignored. This is just like the language you speak. If somebody uses a word that is not in your vocabulary, you don't know what the person is saying to you. Many languages use the same structure (grammar), but different words, and this is the same with XML. Applications that have the same function would benefit from using the same XML vocabulary, just like people benefit from speaking the same language. To make it easier for applications to speak the same language, there are some vocabularies for common applications, some of which are listed here:

- Synchronized Multimedia Integration Language (SMIL), which is used to integrate a set of independent multimedia objects into a synchronized multimedia presentation. See `http://www.w3.org/TR/REC-smil/`.
- Scalable Vector Graphics (SVG), which is a language used to describe two-dimensional vector and mixed vector/raster graphics in XML. See `http://www.w3.org/TR/2001/REC-SVG-20010904/`.
- Simple Object Access Protocol (SOAP), which is used for communication between two systems. Most often the *server* executes some function based on the information transmitted to it and sends back a response. See `http://www.w3.org/TR/SOAP/`.
- Web Services Description Language (WSDL), which is basically used as a sort of directory for finding services (such as SOAP) on a network (specifically the Internet). See `http://www.w3.org/TR/wsdl`.
- XML-based User Interface Language (XUL), which is used to define user interfaces in a Web browser. See `http://www.mozilla.org/xpfe/xptoolkit/xulintro.html`.
- XML Metadata Interchange (XMI), which is an effort to combine the Unified Modeling Language (UML) defined by OMG (`http://www.omg.org`) and XML. See `http://www-4.ibm.com/software/ad/library/standards/xmi.html`.

Many more XML vocabularies are out there and under development for all kinds of applications; however, the listed vocabularies are more visible and widely available because they are endorsed by the W3 Consortium, Mozilla, or the Object Management Group (OMG). A vocabulary that you are by now very familiar with, but which is not in the preceding list, is XSLT. You also can define your own vocabularies using DTDs or XML Schemas.

Vocabulary Collisions

When you have different vocabularies, you might have elements or attributes that have the same name as used in another vocabulary. When these elements or attributes have a different meaning, this can be a cause for problems. For example, in some of the samples used in this book, the element `model` is used to denote a car model:

```
<model name="Focus" manufacturer="Ford" year="2000" />
```

However, the following `model` element has a *totally* different meaning:

```
<model name="Cindy Crawford" height="5'9-1/2" nationality="USA" />
```

Still, both of the preceding elements are well-formed XML. The only way you might be able to tell them apart is by their ancestor element(s) and by the values of the attributes and attribute names. The problem is that elements may even have the same attributes, and as soon as you start checking the values, you bring semantic information into your stylesheets. In addition, your stylesheet can work only with the selected set of values, and any new values are considered as elements from a different vocabulary. The bottom line is that if there is no way to tell apart the vocabularies, you end up with your stylesheet producing the wrong output.

Using Namespaces

Namespaces solve the problem of colliding vocabularies. A namespace consists of two parts: a namespace name and a namespace prefix. The namespace prefix is a prefix added to an element or attribute name. The namespace prefix and element or attribute name are separated by a colon—for instance, `xsl:template`, `car:model`, and `fashion:model`. As you can see from the last two elements, a namespace prefix allows you to treat the two `model` elements as different elements.

A namespace can be declared at any point in an XML document; however, it should be declared in the root element of the tree fragment using the namespace, or in any of the ancestor elements of that root element. In an XSLT document, it therefore needs to be declared in the root element of the stylesheet, as the namespace is being used already by the root element itself, either by `xsl:stylesheet` or the alternative root element `xsl:transform`. For exactly this reason, the root element in XSLT always looks like this:

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

The `xmlns` attribute denotes a namespace declaration, with the namespace prefix being used following the colon (in this case, `xsl`). The attribute's value is the namespace name being used. The namespace name should uniquely identify the namespace. If in two different documents a namespace is declared with the same namespace name, the elements and attributes in that namespace are the same. It doesn't matter whether the namespace prefix used in the different documents is different. An XML parser or processor treats different prefixes as the same namespace when their namespace name is the same. Therefore, Listing 15.1 and Listing 15.2 are identical.

LISTING 15.1 Stylesheet Using `xsl` Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

LISTING 15.2 Stylesheet Using `abc` Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<abc:stylesheet version="1.0"
  xmlns:abc="http://www.w3.org/1999/XSL/Transform">

  <abc:template match="/">
    <abc:apply-templates />
  </abc:template>
</abc:stylesheet>
```

ANALYSIS As far as any parser or processor is concerned, Listing 15.1 and Listing 15.2 are *semantically* the same, because although the namespace prefixes `xsl` and `abc` differ significantly, the namespace name used in the namespace declaration is the same. Therefore, they are treated as the same namespace, and `xsl:template` and `abc:template` are the same elements. It is important that you understand the distinction between the namespace, the namespace name, and the prefix used for that namespace within an XML document. The namespace is defined by the namespace identifier; the prefix is just a means to identify the elements and attributes of a namespace within a document.

Tip

Although namespaces are semantically the same when they use the namespace name, using different prefixes for namespaces across documents is not a good idea. It is much better to keep the prefix of a namespace consistent across documents. This way, you can more easily work with them in large projects or even across projects.

The namespace name has to be a unique identifier because you need to be sure that, if the namespace name is used in two documents, the namespace and vocabulary used are actually the same for those documents. This is why the namespace name is defined as a URI because a URI points to a globally unique location and is therefore a globally unique identifier. The URI is commonly a URL pointing to a document or directory on a specific server. This more or less guarantees that the URI is unique and that it is persistent. You would expect the URI to yield a document defining the vocabulary such as a DTD or XML Schema. This, however, is by no means mandatory, and you will often find that, if you use the namespace name in a browser, you will get a `404 Not Found` error because namespace declarations are just namespace declarations by design. The XML Namespace Recommendation (`http://www.w3.org/TR/1999/REC-xml-names-19990114/`) specifically says that it is not the goal of the namespace declaration to retrieve a DTD or Schema. It is reasonable to assume that this also means that the namespace declaration is not meant to be used to validate the XML using the specified namespace. That said, MSXML is one of the parser/processors that supports this functionality anyway.

Namespace URI Discussion

That namespace names are defined using URIs has been the cause of much debate in the XML community. The question is whether the URI should point to anything in particular, such as a DTD, Schema, or some other form of vocabulary definition. Because the namespace declaration is not specifically meant for validation of content, the answer is basically no. However, answering that question doesn't solve the problem: Where should the URI point?

There seems to be a consensus now on using the Resource Directory Description Language (RDDL). The idea is that a namespace identifier should basically point to a directory. This directory can then hold RDDL and other documents like a DTD and/or Schema. It can also hold documents that are more readable for human users—for example, a description of what the namespace was created for and so on. This reflects the idea that the definitions should be readable by different types of clients, and for both computer systems and humans.

The Benefits of Namespaces

The first and foremost task of namespaces is to keep apart different vocabularies. In a single document, using a single vocabulary might not make a lot of sense to you, but consider what happens when you combine documents that may or may not have different vocabularies. If you don't define namespaces in each document, you might get into the same trouble as discussed earlier. The point is that XML resources will likely be a part of a larger system. In fact, on the Internet, it is likely that someone will use data from another XML document, even if that wasn't intended. So, it is useful to think beyond what you're doing and to think beyond what you're doing *now*. Others might want to build on what you have created, or you might want to expand on what you have already done. Without namespaces, you may hit a brick wall somewhere down the road.

Because you can mix vocabularies within one XML document, you can mix information of the different vocabularies as well. The elements and hierarchies in the document do not necessarily have to be of the same namespace. This is handy if you want to keep data together but want to be able to process it separately. Listing 15.3 shows an XML source that uses this concept.

LISTING 15.3 XML Document Mixing Namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<shop:basket xmlns:shop="http://www.example.com/xmlns/shop"
             xmlns:product="http://www.example.com/xmlns/products">
  <product:product product:ID="234" product:description="Bordeaux"
               product:price="50.00" shop:quantity="1"/>
```

```
    <product:product product:ID="123" product:description="Brie"
                 product:price="99.95" shop:quantity="3"/>
</shop:basket>
```

ANALYSIS Listing 15.3 uses two namespaces: one uses the prefix `shop;` and the other, the prefix `product`. The namespaces are used mixed with one another. The benefit of this approach is that the attributes belonging to the `product` namespace can be processed separately if necessary. You can address them as a group by using the expression `@product:*`, so it is not relevant which attributes are there. These data items are all related and stay the same over a period of time. The data in the `shop` namespace is specific to one user and will change often. You could also envision adding the user's address information to the document, possibly using a different namespace. This way, the user's `shopping` information is grouped into one document that can be used while the user is shopping, to do the checkout, for shipping and handling, for invoicing, and so on. Each process needs only part of the information, but it needs to be grouped in some way to be relevant. You don't want to have a separate invoice, a different document for shipping, and so on.

The Drawbacks of Namespaces

Namespaces are handy, but they add to the complexity of a document, specifically when it contains more than one namespace.

Note

In a document, you can define the default namespace by using `xmlns="some URI"`. Elements without a prefix are then of that namespace. This definition is particularly useful in documents that use only one namespace because it cuts down on typing.

A more complex XML document means that it takes longer to create and, even worse, that any XSLT stylesheet also becomes more complex. A stylesheet needs to declare the namespace just like the XML source the stylesheet is operating on. Fortunately, the namespace prefix may be different than that used in the source document. As long as the namespace name is the same, the stylesheet will work fine with the XML source, and the data in the XML source will be treated as if it has the prefix defined in the stylesheet. This way namespaces in two different documents with the same prefix can be treated by the stylesheet as if they have different prefixes. So, if your stylesheet needs to work on both documents, it has a way to distinguish the two namespaces.

NEW TERM When you're mixing vocabularies using namespaces, validating the XML becomes much harder. Although, technically, this situation doesn't have to do with namespaces, but rather with validation, it is very much related. The point is that if you're validating a document that uses multiple namespaces, it is likely that you are validating against multiple DTDs or Schemas. If the validation fails on any one DTD or Schema, the entire operation is canceled and the document is loaded and transformed. This is specifically a problem if you're using closed DTDs or Schemas. A *closed* DTD or Schema cannot be extended upon. Its vocabulary and hierarchy are strictly defined and enforced at validation time.

That a closed DTD or Schema will not work properly isn't really a surprise. Because it is closed, you shouldn't extend it. Mixing it up with other elements from other namespaces to the parser means extending it, so that is illegal by the very nature of the DTD or Schema. Nevertheless, this situation is not uncommon and not the only cause for problems. DTDs and Schemas expect certain elements and attributes in certain places. If they aren't there, validation fails and so does everything else. When you're mixing namespaces, you might be violating rules in a DTD or Schema just because it has an element of another namespace that the parser didn't expect. This means that the parser has no choice but to abort. Unless the parser or processor you're working with gives details about the validation error, tracking it down can be very hard. Also, if you look at the result in Internet Explorer, you might not see an error, but just a document that looks different than it is supposed to.

Namespaces, DTDs, and Schemas

It is very natural to think that namespaces and vocabulary definitions with a DTD or XML Schema are linked together. In fact, they are not. A namespace is just meant to keep vocabularies apart. Whether you validate a source document with a DTD or Schema is of no concern to the namespace, nor is the namespace concerned with validating documents itself.

Validation by a DTD is fairly simple. If a DTD is associated with an XML source using a `DOCTYPE` declaration, a validating parser will validate the document when it is loaded. Most parsers/processors implement this behavior.

XML Schema is fairly new, so you should check whether the parser/processor you use supports validation against a Schema. The parser that currently comes with Saxon, for instance, doesn't support XML Schema; MSXML 4.0 does, however. MSXML 3.0, which was released before the XML Schema Recommendation, supports a mechanism to validate against a Schema that is different from the mechanism defined in the XML Schema Recommendation. An example of validation following the recommendation is shown in Listing 15.4.

LISTING 15.4 Validation Information for XML Schema

```
1: <car:model
2:   xmlns:car="http://www.example.com/xmlns/car"
3:   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4:   xsi:schemaLocation="http://www.example.com/xmlns/car
5:   http://www.example.com/xmlns/car/car.xsd"
6:   name="Golf" manufacturer="Volkswagen" year="1999" />
```

ANALYSIS The document in Listing 15.4 has a single element. This point is not very significant, but the namespace declarations and their use are. On line 2, the namespace for the `car` prefix is declared. On line 3, the `xsi` prefix is declared for the namespace name `http://www.w3.org/2001/XMLSchema-instance`. This namespace is the one used for XML Schema validation. Just like the namespace used for XSLT, this namespace must be associated with the given URI, or it will not be regarded as the XML Schema validation namespace. This namespace is used on lines 4 and 5 to define the Schema that should be used to validate the XML document. Where the Schema is located is defined using the `xsi:SchemaLocation` attribute, which first holds the namespace name (line 4) and then the location of the Schema to be associated with that namespace name (line 5). The two URIs are separated by whitespace. If you're working with an XML Schema–compliant parser/processor, such as MSXML 4.0, Listing 15.4 will be validated against the Schema located at `http://www.example.com/xmlns/car/car.xsd`.

Note The `car.xsd` Schema is not provided here because Schemas are beyond the scope of this book. Schemas are discussed here in the context of their relationship with namespaces.

In MSXML 3.0 and higher, you also can validate against an XML Schema by adding `x-Schema` in front of the namespace name, like this:

```
xmlns:car="x-Schema:http://www.example.com/xmlns/car/car.xsd"
```

The drawback of this method is that the namespace name and the Schema location are one and the same, which, as explained earlier, is not the best solution.

Caution Validation information should be incorporated *only* in the source document. A common mistake, particularly with the MSXML Schema validation method, is to include this information in the stylesheet namespace declaration as well. The result is that your stylesheet is not valid and that, at the very least, your output will be incorrect.

Processing XML Sources with Namespaces

In the preceding section, you learned all the theory surrounding namespaces. The next step is, of course, learning about actually using namespaces. The first thing you need to know is how to process documents containing one or more namespaces. Listing 15.5 is an example of such a document.

LISTING 15.5 Sample XML Source with Namespace

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <car:cars xmlns:car="http://www.example.com/xmlns/car">
3:    <car:model name="Golf" manufacturer="Volkswagen" year="1999" />
4:    <car:model name="Camry" manufacturer="Toyota" year="1999" />
5:    <car:model name="Focus" manufacturer="Ford" year="2000" />
6:    <car:model name="Civic" manufacturer="Honda" year="2000" />
7:    <car:model name="Prizm" manufacturer="Chevrolet" year="2000" />
8:    <car:model name="Celica" manufacturer="Toyota" year="2000" />
9:    <car:model name="Mustang" manufacturer="Ford" year="2001" />
10:   <car:model name="Passat" manufacturer="Volkswagen" year="2001" />
11:   <car:model name="Accord" manufacturer="Honda" year="2002" />
12:   <car:model name="Corvette" manufacturer="Chevrolet" year="2002" />
13: </car:cars>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS Listing 15.5 contains an XML source using one namespace. The namespace defined by the `http://www.example.com/xmlns/car` namespace name appears on line 2. In this document, it is given the prefix `car`. Because the root element itself uses the namespace, the namespace needs to be declared as part of the root element. Otherwise, the namespace could be declared when it is first used in the document. You can always declare all namespaces in the root element if you like. Note that the element name uses the namespace, but the attributes do not. This fact is very important because it means that no namespace is associated with the attributes (unless a default namespace has been declared). Any template matching attributes with the `car` namespace will not match the attributes in Listing 15.5.

If you want to associate a namespace with attributes, you have to add a namespace prefix as well, as shown in Listing 15.3 earlier. Unless you're sure that your attributes are either unique or the same as attributes with the same name in other vocabularies, you should add a namespace to the attributes as well, although doing so makes your XML source larger (and more work to type). As I said earlier, you can declare a default namespace to be associated with elements and attributes that use no prefix.

When you have an XML source using a namespace, your stylesheet needs to process the elements as part of that namespace. This means that it needs to have knowledge of that namespace and possibly process nodes according to their namespace. To give the stylesheet knowledge about a namespace, you have to declare that namespace in the stylesheet, just as you did in the source document, as shown in Listing 15.6.

LISTING 15.6 Stylesheet Using Namespace from Listing 15.5

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:auto="http://www.example.com/xmlns/car">
5:
6:    <xsl:output method="text" encoding="UTF-8" />
7:    <xsl:strip-space elements="*" />
8:
9:    <xsl:template match="/">
10:     <xsl:apply-templates />
11:   </xsl:template>
12:
13:   <xsl:template match="auto:cars">
14:     <xsl:apply-templates />
15:   </xsl:template>
16:
17:   <xsl:template match="auto:model">
18:     <xsl:value-of select="@manufacturer" />
19:     <xsl:text> </xsl:text>
20:     <xsl:value-of select="@name" />
21:     <xsl:text> (</xsl:text>
22:     <xsl:value-of select="@year" />
23:     <xsl:text>)&#xA;</xsl:text>
24:   </xsl:template>
25: </xsl:stylesheet>
```

ANALYSIS Because Listing 15.6 operates on an XML source using namespaces, it needs to be aware of those namespaces. In this case, only one namespace is used, and it is declared on line 4. Note that the namespace name is the same as in Listing 15.5, but the prefix is different. The processor will translate the prefix used in the source XML to the prefix used in the stylesheet. This way, if another document is used with another namespace but the same prefix, it can just be defined with another prefix in the stylesheet, so they can't be in each other's way. The elements that are associated with the namespace in Listing 15.5 are matched on lines 13 and 17 by two different templates. As you can see, those templates use the namespace prefix declared on line 4. The attributes of the `model` element, on the other hand, are selected without using a namespace prefix because they aren't associated with a namespace. If line 20 used `@auto:name`, it would select nothing because that attribute is not available, which shows you that there really is no namespace associated with the attribute. The output of Listing 15.6 is shown in Listing 15.7.

OUTPUT **LISTING 15.7** Result from Applying Listing 15.6 to Listing 15.5

```
Volkswagen Golf (1999)
Toyota Camry (1999)
Ford Focus (2000)
Honda Civic (2000)
Chevrolet Prizm (2000)
Toyota Celica (2000)
Ford Mustang (2001)
Volkswagen Passat (2001)
Honda Accord (2002)
Chevrolet Corvette (2002)
```

ANALYSIS Although Listing 15.6 works with namespaces, the output in Listing 15.7 is just plain text, which is familiar from earlier lessons.

Getting Namespace Information

When you're using namespaces, you can get information about the namespace of an element in different ways. First, you can get the namespace prefix of the current element by using the `substring-before()` function. So, the following line of code would yield the prefix of the current element:

```
substring(.,':')
```

You also can very easily check whether elements are of a certain namespace by using the following line of code:

```
<xsl:if test="self::car:*" xmlns:car="http://www.example.com/xmlns/car">
```

The body of the preceding code would be executed if the current element is part of the `http://www.example.com/xmlns/car` namespace. Note that the namespace is declared in the `xsl:if` element instead of in the root element. If you just want to get the namespace name of the namespace associated with the current element, you can also use the `namespace-uri()` function, which returns the URI that is used as the namespace name. Because you can test the namespace of an element just by using its prefix, this function is not very useful in most stylesheets. However, when you're doing some kind of reporting or testing, this function may be useful. Listing 15.8 uses this function.

LISTING 15.8 Stylesheet Using the `namespace-uri()` Function

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:auto="http://www.example.com/xmlns/car">
5:
6:    <xsl:output method="text" encoding="UTF-8" />
7:
8:    <xsl:template match="/">
9:      <xsl:value-of select="namespace-uri(child::*)" />
10:   </xsl:template>
11: </xsl:stylesheet>
```

ANALYSIS The code in Listing 15.8 does nothing more than output the namespace name (URI) of the root node of the document being processed. It works only on another stylesheet or a document having a root node associated with the `http://www.example.com/xmlns/car` namespace because only these namespaces are declared in this stylesheet on lines 3 and 4. Line 9 uses the `namespace-uri()` function to output the namespace name of the root element, selected using the `child::*` expression, which selects any child node of the document root. If the root element of the source document is associated with the `http://www.example.com/xmlns/car` namespace, that namespace is the output. Otherwise, the result is empty.

Inserting and Removing Namespaces

In the preceding sections, you learned how to work with a document that uses namespaces. As yet, you can use only documents that use namespaces. Creating a document that uses a namespace or changing the namespace that you got from the output is a different matter.

Inserting Nodes with Namespaces

Inserting namespaces into the output document is remarkably easy. You can just insert the node with the namespace prefix and colon, and you're done. The only requirement is that the namespace is declared within the stylesheet. It is a good idea to declare all the namespaces in the root element of the stylesheet so that you always can quickly see which namespaces have been declared and with which prefix. A nice way to have a look at inserting namespaces is to transform an XML source that doesn't have namespaces into one that does have them. Listing 15.9 shows the XML source to be transformed.

LISTING 15.9 XML Source Without a Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car name="Golf" manufacturer="Volkswagen" year="1999" />
  <car name="Camry" manufacturer="Toyota" year="1999" />
  <car name="Focus" manufacturer="Ford" year="2000" />
  <car name="Civic" manufacturer="Honda" year="2000" />
  <car name="Prizm" manufacturer="Chevrolet" year="2000" />
</cars>
```

Listing 15.9 is one of the familiar variants of car information documents used throughout this book. Now, however, it will be changed into a document that uses a namespace for the elements and attributes. Listing 15.10 is the stylesheet that will transform Listing 15.9.

LISTING 15.10 Stylesheet Creating a Document with a Namespace

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:car="http://www.example.com/xmlns/car">
5:
6:    <xsl:output method="xml" encoding="UTF-8" indent="yes" />
7:    <xsl:strip-space elements="*" />
8:
9:    <xsl:template match="/">
10:     <xsl:apply-templates />
11:   </xsl:template>
12:
13:   <xsl:template match="cars">
14:     <car:cars>
15:       <xsl:apply-templates />
16:     </car:cars>
17:   </xsl:template>
18:
19:   <xsl:template match="car">
20:     <car:car>
21:       <xsl:apply-templates select="@*" />
22:     </car:car>
23:   </xsl:template>
24:
25:   <xsl:template match="@*">
26:     <xsl:attribute name="car:{name()}">
27:       <xsl:value-of select="." />
28:     </xsl:attribute>
29:   </xsl:template>
30: </xsl:stylesheet>
```

ANALYSIS Listing 15.10 will transform Listing 15.9 into a document using a namespace (shown in Listing 15.11). Because the stylesheet basically copies the existing nodes, it doesn't have a lot of exotic functionality. The template matching the `cars` element on line 13 inserts the element into the output as `car:cars` on line 14. This is possible because that namespace is declared on line 4 of the stylesheet. If it weren't, line 14 would generate an error. The same thing is done for the `car` elements in the template starting on line 19. On line 21, this same template initiates processing for all the attributes of the `car` element. The attributes are processed by the template on line 25. Line 26 inserts the attribute, preceded by the namespace prefix. As you can see, the name is constructed from the prefix and a dynamic part using the `name()` function to get the original name of the attribute from the source document.

OUTPUT **LISTING 15.11** Result from Applying Listing 15.10 to Listing 15.9

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <car:cars xmlns:car="http://www.example.com/xmlns/car">
3:   <car:car car:name="Golf" car:manufacturer="Volkswagen" car:year="1999"/>
4:   <car:car car:name="Camry" car:manufacturer="Toyota" car:year="1999"/>
5:   <car:car car:name="Focus" car:manufacturer="Ford" car:year="2000"/>
6:   <car:car car:name="Civic" car:manufacturer="Honda" car:year="2000"/>
7:   <car:car car:name="Prizm" car:manufacturer="Chevrolet" car:year="2000"/>
8: </car:cars>
```

ANALYSIS Listing 15.11 uses the namespace that was inserted in Listing 15.10. As you can see, the namespace is declared on line 2 and then used for each element and attribute.

Changing Namespaces

With XSLT, you can change the representation of a namespace as used in a stylesheet to something different for use in the output. At first glance, changing the namespace seems totally unnecessary, but think about creating stylesheets with a stylesheet. If you want to output an XSLT element like `xsl:stylesheet`, you can't insert it into the stylesheet as such. If you did that, it would be interpreted as an XSLT element in the stylesheet, and not as an element to be sent to the output. The solution is to use a temporary alias for the XSLT elements you want to output. If you use `axsl:stylesheet` and declare a separate namespace for it, the stylesheet doesn't have a problem because it can make a distinction between `xsl` and `axsl`.

The preceding approach still poses one problem, however. The namespace associated with the alias must be different in the stylesheet, which means that it also will be different in the output. Because XSLT relies on the namespace name to identify it as XSLT, the namespace associated with the output would not identify it as XSLT. This is where the `xsl:namespace-alias` element comes in. It allows you to map a temporary alias to another namespace in the stylesheet. When the output is generated, the namespace prefix is substituted for the one used in the stylesheet, but more important, the namespace name associated with it is substituted as well. Listing 15.12 shows how to use a namespace alias.

LISTING 15.12 Stylesheet Using a Namespace Alias

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:out="temp">
5:
6:    <xsl:output method="xml" encoding="UTF-8" indent="yes" />
7:    <xsl:strip-space elements="*" />
8:    <xsl:namespace-alias result-prefix="xsl" stylesheet-prefix="out" />
9:
10:   <xsl:template match="/">
11:     <out:stylesheet version="1.0">
12:       <out:value-of select="." />
13:     </out:stylesheet>
14:   </xsl:template>
15: </xsl:stylesheet>
```

ANALYSIS Listing 15.12 creates another stylesheet. To do that, a temporary namespace is declared on line 4. This listing uses `out` as a namespace prefix and `temp` as a namespace name, but they could be nearly anything, as long as they aren't the same as any other namespace declarations in the stylesheet. Line 8 maps the `out` namespace to the `xsl` namespace. The `result-prefix` attribute defines the namespace to be used in the output, and the `stylesheet-prefix` attribute defines the namespace used within the stylesheet. When the output is created, the `xsl` namespace prefix is used, along with the namespace name used on line 3. On lines 11 through 13, elements are inserted using the `out` namespace prefix. Note that the elements inserted are valid XSLT elements with the correct attributes. When the output is created (using any XML source), the result should look like Listing 15.13.

OUTPUT **LISTING 15.13** Result from Listing 15.12

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
 ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:value-of select="." />
</xsl:stylesheet>
```

Analysis Listing 15.13 is the exact stylesheet that was created by Listing 15.12, containing the elements inserted. Because of the `xsl:namespace-alias` element, the namespace prefix used is `xsl`. In addition, line 2 shows that the namespace name used is the URI needed to associate the `xsl` prefix with XSLT, so a processor will actually regard Listing 15.12 as a stylesheet. The temporary namespace declared in Listing 15.12 is nowhere to be found, which is as it should be.

The behavior for the different processors is not the same. The XSLT specification states that the only requirement is that the namespace name is correct, although the intention is likely to be as shown in Listing 15.13. MSXML produces this output, but Saxon and Xalan produce different output from MSXML and each other. The output for Saxon is shown in Listing 15.14 and for Xalan in Listing 15.15.

Output **Listing 15.14** Result from Listing 15.12 with Saxon

```
<?xml version="1.0" encoding="UTF-8"?>
<out:stylesheet xmlns:out="http://www.w3.org/1999/XSL/Transform"
➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <out:value-of select="."/>
</out:stylesheet>
```

Output **Listing 15.15** Result from Listing 15.12 with Xalan

```
<?xml version="1.0" encoding="UTF-8"?>
<out:stylesheet xmlns:out="http://www.w3.org/1999/XSL/Transform"
➥ version="1.0">
<out:value-of select="."/>
</out:stylesheet>
```

Analysis Listings 15.14 and 15.15 are just as valid as Listing 15.13. Both use the `alias` prefix in Listing 15.12 for the output and attach the original namespace to that prefix. Because the namespace name defines the namespace, and not the prefix, the output is semantically the same. In addition, Saxon adds the original namespace, although this is completely superfluous.

Note

Listings 15.13, 15.14, and 15.15 are the same for all intents and purposes. Don't be fooled by the different appearances. Other processors may produce even different output, but the output is always semantically the same.

The sample in Listing 15.12 is, of course, not very practical because it creates a stylesheet from a stylesheet and does nothing to change the resulting stylesheet according to some source document. It does, however, serve its purpose as a short example with a namespace alias. You can use the same method to create a stylesheet based on some XML source.

Removing Namespaces

If you declare a namespace in a stylesheet, that namespace declaration is copied to the output if you're generating XML or HTML. This also is the case if a namespace isn't even used in the output document. The only exception is the XSLT namespace, which is not sent to the output, unless you have used `xsl:namespace-alias` to alias the XSLT namespace.

Namespace declarations usually don't bite when they are included in the output but not used. It doesn't look pretty, though, if a document is processed a few times, and each time a new, not used, namespace declaration is added. Fortunately, you can remove namespace declarations by using the `exclude-result-prefixes` attribute of the `xsl:stylesheet` element. Its value needs to be a whitespace-separated list of prefixes of the namespaces you want to exclude from the output. Any namespace that is not used in the output is completely omitted from it, including its declaration. If a namespace is used in the output and is listed in the `exclude-result-prefixes` attribute, the namespace is declared in the output anyway. The most common use for this attribute is creating HTML documents, which Listing 15.16 demonstrates.

LISTING 15.16 Stylesheet Creating an HTML Document with a Table from Listing 15.11

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="http://www.example.com/xmlns/car">

  <xsl:output method="html" encoding="UTF-8" indent="yes" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <html>
    <body>
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="car:cars">
    <table>
```

```
19:       <xsl:apply-templates />
20:     </table>
21:   </xsl:template>
22:
23:   <xsl:template match="car:car">
24:   <tr>
25:     <xsl:apply-templates select="@*" />
26:   </tr>
27:   </xsl:template>
28:
29:   <xsl:template match="@*">
30:     <td><xsl:value-of select="." /></td>
31:   </xsl:template>
32: </xsl:stylesheet>
```

ANALYSIS Listing 15.16 creates an HTML document with a table listing the `car:car` elements in Listing 15.11. Essentially, you have done the same thing before, except for the namespace used in Listing 15.11 and declared on line 4 of the stylesheet. Listing 15.16 yields the result in Listing 15.17.

OUTPUT **LISTING 15.17** Result from Applying Listing 15.16 to Listing 15.11

```
<html xmlns:car="http://www.example.com/xmlns/car">
   <body>
      <table>
         <tr>
            <td>Golf</td>
            <td>Volkswagen</td>
            <td>1999</td>
         </tr>
         <tr>
            <td>Camry</td>
            <td>Toyota</td>
            <td>1999</td>
         </tr>
         <tr>
            <td>Focus</td>
            <td>Ford</td>
            <td>2000</td>
         </tr>
         <tr>
            <td>Civic</td>
            <td>Honda</td>
            <td>2000</td>
         </tr>
         <tr>
            <td>Prizm</td>
            <td>Chevrolet</td>
```

LISTING 15.17 Continued

```
            <td>2000</td>
         </tr>
      </table>
   </body>
</html>
```

ANALYSIS Listing 15.17 is just a plain HTML document. The only exception is that the first line contains a namespace declaration for the `car` prefix. Clearly, that namespace has no purpose because the browser ignores it and just displays the HTML. If you added `exclude-result-prefixes="car"` to the `xsl:stylesheet` element in Listing 15.16, the namespace declaration in Listing 15.17 would not be there, resulting in a perfect HTML document.

Removing Namespace Prefixes

Earlier you learned how to create a document using namespaces from one that didn't. The last subject of this lesson goes the other way around, removing namespaces and matching nodes on just their name, without the namespace prefix. Basically, you can choose from two methods to remove namespace prefixes. The first method uses the `substring-after()` function to get the name of a node without the namespace prefix. You can, however, use an easier method, so this method will not be discussed further. Using the `local-name()` function, you can get the name of a node without the prefix, and as such, this function is also suited to do name tests in expressions. This capability is useful if you have nodes in different namespaces that need to be matched by the same template, although that contradicts the issue of separating vocabularies using namespaces. You can imagine that different namespaces use some of the same elements and attributes, such as names. When that happens, you can use the same template to process these elements, as shown by the example in Listing 15.18.

LISTING 15.18 Sample XML Source with Multiple Namespaces

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <car:cars xmlns:car="http://www.example.com/xmlns/car"
3:            xmlns:m="http://www.example.com/xmlns/manufacturer">
4:    <car:models>
5:      <car:model car:name="Golf" m:id="VW" car:year="1999" />
6:      <car:model car:name="Camry" m:id="TY" car:year="1999" />
7:      <car:model car:name="Focus" m:id="FO" car:year="2000" />
8:      <car:model car:name="Civic" m:id="HO" car:year="2000" />
9:      <car:model car:name="Prizm" m:id="CV" car:year="2000" />
10:   </car:models>
11:   <m:manufacturers>
```

```
12:     <m:manufacturer m:id="VW" m:name="Volkswagen" m:country="Germany" />
13:     <m:manufacturer m:id="TY" m:name="Toyota" m:country="Japan" />
14:     <m:manufacturer m:id="FO" m:name="Ford" m:country="USA" />
15:     <m:manufacturer m:id="CV" m:name="Chevrolet" m:country="USA" />
16:     <m:manufacturer m:id="HO" m:name="Honda" m:country="Japan" />
17:   </m:manufacturers>
18: </car:cars>
```

ANALYSIS Listing 15.18 mixes two namespaces: one for cars and one for manufacturers. These namespaces are declared on lines 2 and 3, and are used by both elements and attributes. Because the manufacturers are related to the cars, the common root element is of the `car` namespace. Note that the elements have the `name` attribute in common. Listing 15.19 uses this knowledge.

LISTING 15.19 Stylesheet Using the `local-name()` Function

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="http://www.example.com/xmlns/car"
  xmlns:man="http://www.example.com/xmlns/manufacturer">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

  <xsl:template match="car:models">
    <xsl:text>Car models:&#xA;</xsl:text>
    <xsl:apply-templates select="car:model/@car:name" />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="man:manufacturers">
    <xsl:text>Manufacturers:&#xA;</xsl:text>
    <xsl:apply-templates select="man:manufacturer/@man:name" />

  </xsl:template>

  <xsl:template match="@*[local-name()='name']">
    <xsl:value-of select="." />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 15.19 operates on Listing 15.18, so it needs both namespaces declared in Listing 15.18 declared as well. These declarations appear on lines 4 and 5. Listing 15.19 will create a text document that consists of a list of cars and a list of manufacturers. Both have a `name` attribute, and the template starting on line 25 will match both of them. In fact, the match expression will match any `name` attribute in any namespace because it uses the `local-name()` function and compares the result with a string. You also can use the `local-name()` function on different nodes by passing a select expression as the argument. The result is shown in Listing 15.20.

OUTPUT **LISTING 15.20** Result from Applying Listing 15.19 to Listing 15.18

```
Car models:
Golf
Camry
Focus
Civic
Prizm

Manufacturers:
Volkswagen
Toyota
Ford
Chevrolet
Honda
```

Summary

In today's lesson, you learned that you can use namespaces to keep apart vocabularies. Namespaces are denoted by a prefix that can be used in front of an element or attribute name. The name and the namespace prefix are separated by a colon. To be able to use a certain prefix, you need to declare it first in or before the outermost element using the prefix. It is a good idea to declare all namespaces in the root of a document rather than scatter namespace declarations throughout a document.

Although namespaces are related to DTDs and Schemas, the namespace declaration is not meant to point to a DTD or Schema to get the vocabulary for a certain namespace. The validation process of a source document and the namespace declarations are separate functionalities.

In cases in which you need to output a namespace that is used as part of XSLT (or another XML language), you can define a namespace alias for that namespace. You can use another prefix so that the nodes will be seen as output, not as XSLT elements to be processed.

In tomorrow's lesson, you'll learn more details about selecting data. You'll revisit and examine further some concepts you learned in the first week, and learn some additional techniques to select data.

Q&A

Q Is there a limit to the number of namespaces I can use in a document and a stylesheet?

A No. You can use any number of namespaces. Your documents will, however, not become more readable if they have many namespaces. You should probably make a different document design.

Q Can I alias the default namespace?

A Yes. You can use `#default` as a pseudo prefix for the default namespace.

Q Why isn't there a function that returns just the namespace prefix?

A There is really no need for such a function. You can address elements in the same namespace by using `prefix:*`. If you really need only the namespace prefix, use `substring-before(name(),':')`.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Attributes in an element can have different namespaces.
2. True or False: The namespace prefix in a stylesheet needs to be the same as that of the source document.
3. Why isn't a namespace name supposed to point to a DTD or an XML Schema?
4. Why doesn't an expression matching a specific element name also match elements in a namespace with the same name?
5. How can you make sure that a namespace is not declared in the output if that namespace isn't used in the output?

Exercise

1. The following stylesheet comes from Day 8, “Working with Variables.” It creates an HTML document for an auto show, with the manufacturers listed with their cars. Change the stylesheet so that it works with Listing 15.18. Make sure that the namespace declarations are not carried over to the output.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates select="/cars/manufacturers" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:variable name="mfc" select="." />
      <xsl:for-each select="/cars/models/model[@manufacturer = $mfc/@id]">
        <ul>
          <li>
            <xsl:value-of select="@name" />
            <xsl:text> (</xsl:text>
            <xsl:value-of select="@year" />
            <xsl:text>)</xsl:text>
          </li>
        </ul>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

DAY 16

Advanced Data Selection

Yesterday you learned that using namespaces is a great way to structure documents and separate colliding vocabularies. Although namespaces make your documents more complex, they also keep you from processing elements of different vocabularies the same way, which is often not what you want.

Today's lesson takes a closer look at selecting data with expressions. This topic was covered in the first week, but there you only scratched the surface. With each lesson since then, your understanding of expressions and data selection has grown. Now it's time to go back to them for the finishing touch. Today you will learn more details about building based on your current knowledge and will learn about some more selection techniques available to you.

In today's lesson, you will learn the following:

- How to use implicit data type conversions to your advantage
- How to select unique values from a node-set with duplicate values
- How you can use keys to access data quickly
- How to work with unique ID values

More About Expressions

Expressions were discussed on Day 3, “Selecting Data,” and have been used throughout this book. So, why come back to them now? Expressions and patterns are the most important part of XSLT because they define which data you are actually working with. Without expressions and patterns, XSLT wouldn’t be worth anything. In this section, I will discuss some characteristics of expressions. You have been introduced implicitly to most of these characteristics, but it is useful to make you fully aware of them.

There are basically two types of expressions: one used to match or select some data, the other to compare data. Of course, these two are mixed in many cases, because when you compare data, you have to select data as well. Also, when you select data, you can use predicates that make a comparison to select certain data.

To discuss expressions, this section uses the familiar source document shown in Listing 16.1.

LISTING 16.1 Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Ceasar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.125">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.125">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note

You can download the sample listings in this lesson from the publisher’s Web site.

Matching and Selecting Data

Let's look first at match and select expressions without predicates that make comparisons. Basically, such an expression is a location path. The result of that location path depends on how specific the location path is. For instance, the expression `/menu/entrees/dish[1]/text()` is either empty or a string. The same goes for any location path that selects an attribute of a specific element in a source document—for instance, `/menu/entrees/@title`. Working with expressions starts to become tricky when more elements match your expression, such as `/menu/entrees/dish/text()` or `/menu/*/@title`. Although these expressions point to a text value, the location paths before the `text()` function and attribute selection match multiple nodes and therefore do not return a string, but a node-set.

For the second example, that result is more or less clear because it uses a wildcard. The first one is less obvious, however, because each element is spelled out completely. For a match expression, that isn't such a problem. After all, you created a template to deal with nodes matching the expression, no matter how many. When you're selecting data, it can be a problem, though, because you are expecting a string, but a node-set is returned. Still, the expression will not fail, nor will the entire code. If a string is expected, either by a function or a comparison, the node-set is converted implicitly to a string. As discussed on Day 10, "Understanding Data Types," this results in the value of the first node being used and converted to a string if necessary. This means that `/menu/entrees/dish[1]/text()` and `/menu/entrees/dish/text()` actually yield the same string result, even though the former results in a string and the second in a node-set. When you create expressions, you are bound to run into such implicit conversions.

Comparing Values

Implicit conversion isn't all bad. In fact, it can be of great benefit because you can quickly check whether a value exists. So, especially in comparisons, implicit conversion can be quite useful. Whether you actually use implicit conversion in comparisons is a matter of choice. If you don't want to run into any unforeseen trouble, converting every value explicitly is the better choice. The downside is that doing so probably decreases the performance of your stylesheet because implicit conversion is bound to be more efficient.

A great way to make use of implicit conversion is in test expressions of `xsl:if` and `xsl:when` elements. If you want to check whether some node or nodes exist, you can use the following code:

```
<xsl:if test="/menu/desserts/dish">
```

This test expression looks a little weird, but it actually tests to see whether there are any desserts. If the `desserts` element has no `dish` child elements, the node-set returned by the expression is empty. This is implicitly converted to a Boolean value, which is the data type the `test` attribute expects. When this expression yields an empty node-set, the converted value is `false`; otherwise, it is `true`. This means that if there are any desserts, the code block inside the `xsl:if` element is processed; otherwise, it is not.

Another useful feature of comparisons is that you can compare a simple value, such as a Boolean value, number, or string, to a node-set. If the node-set contains a node that has the value you require, the expression returns `true`. So, the comparison `/menu/*/dish = 'Ribs and Wings'` returns `true` for Listing 16.1.

Selecting Distinct Values

In documents where elements have multiple attributes, it is not uncommon that the values of some of those attributes are the same. Hence, a selection of such an attribute's node-set would yield a node-set with nondistinct values because some of the values would be duplicated once or more in the node-set. Listing 16.2 should make this clearer.

LISTING 16.2 XML Source with Duplicate Manufacturers

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <model name="Golf" manufacturer="Volkswagen" year="1999" />
  <model name="Camry" manufacturer="Toyota" year="1999" />
  <model name="Focus" manufacturer="Ford" year="2000" />
  <model name="Civic" manufacturer="Honda" year="2000" />
  <model name="Prizm" manufacturer="Chevrolet" year="2000" />
  <model name="Celica" manufacturer="Toyota" year="2000" />
  <model name="Mustang" manufacturer="Ford" year="2001" />
  <model name="Passat" manufacturer="Volkswagen" year="2001" />
  <model name="Accord" manufacturer="Honda" year="2002" />
  <model name="Corvette" manufacturer="Chevrolet" year="2002" />
</cars>
```

ANALYSIS In Listing 16.2, each `model` element is unique because the combination of all the attributes for each element is unique. If you look closely at all the `manufacturer` attributes, you'll notice that each manufacturer is listed twice. This means that the expression `/cars/model/@manufacturer` yields a node-set with 10 nodes, but only 5 distinct manufacturers. So, if you list all the manufacturers, you get a list with 10 manufacturers, with each one duplicated once. If you want to list only the manufacturers, you probably don't want those duplicated values, so you need a way to select only the distinct values of the `manufacturer` attribute. Listing 16.3 shows how you can select the distinct values.

LISTING 16.3 Stylesheet Selecting Distinct Manufacturers

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:template match="/">
8:      <xsl:for-each select="//model[not(@manufacturer =
9:           preceding-sibling::model/@manufacturer)]/@manufacturer">
10:       <xsl:sort select="." />
11:       <xsl:value-of select="concat(.,'&#xA;')" />
12:     </xsl:for-each>
13:   </xsl:template>
14: </xsl:stylesheet>
```

ANALYSIS Listing 16.3 is very simple. The template on line 7 matches the root of the source document. This template iterates through the manufacturers with the `xsl:for-each` element on line 8. For each manufacturer, the value is written to the output. Line 10 sorts the manufacturers before sending them to the output. The most interesting thing about Listing 16.3 is the `select` expression of the `xsl:for-each` element on line 8. This expression spans two lines. The expression selects all the `model` elements in the source document and uses a predicate to filter out elements with duplicate `manufacturer` attributes. After the filter is applied, the `manufacturer` attribute is selected for each node that passed the filter. The filter expression checks whether the value of the `manufacturer` attribute of the current element is equal to the value of a `manufacturer` attribute of any of the preceding `model` elements. If it is equal, the manufacturer has already been sent to the output, so the current one is a duplicate. The `not()` function makes sure that the predicate returns `false` so that the duplicate value isn't sent to the output.

When line 5 of Listing 16.2 is evaluated, the value `Ford` is compared to the values `Toyota` from line 4 and `Volkswagen` from line 3. Both are different, so `Ford` is sent to the output. The `Ford` value on line 9 is compared to the values of the manufacturer attributes on lines 3–8. Because line 5 already has a value `Ford`, the expression `@manufacturer = preceding-sibling::model/@manufacturer` yields `true`, so the `model` element on line 9 is completely ignored. That Listing 16.3 yields only a unique list of manufacturers is shown in Listing 16.4.

OUTPUT **LISTING 16.4** Result from Applying Listing 16.3 to Listing 16.2

```
Chevrolet
Ford
Honda
Toyota
Volkswagen
```

Getting the duplicate values out of Listing 16.2 is a matter if using the right axis. You'll find that when you have to compare values with values of other nodes, axes can do a lot that you can't do with comparisons of values because the current context gets in the way.

Note Familiarizing yourself again with axes is a good idea. Table 3.1 and Figure 3.5 in Day 3's lesson show a reference that you can use when creating expressions that might benefit from using axes.

Working with Keys

In the preceding section, I reminded you that expressions can be quite long if you have to get to specific data. In addition, processors take longer to process a document if expressions are complex. Keys can help to simplify expressions and speed up processing.

What Is a Key?

A key is like an index in a book. In the index, you can look up a word. If that word exists in the book, the index tells you on which page or pages you can find it. A key is similar in that you can look up a value. If the value exists in the key index, you get a node-set containing the node or nodes that have the given value.

Before you can use an index in a book, it has to be compiled. The same is true for a key index in that it needs to be defined before you can use it. You have to define which nodes are to be indexed and the value or values that need to be used for the index. The nodes correspond to the pages in a book; the value or values, to the words found in the index. That you have to define the index is quite logical, because in a book you don't want to index every single word, but just those words that relate to the subject matter. The value or values used to index the nodes can be the value of the nodes themselves, attribute values of the element's attributes, or values of child elements.

When you have an index, you can look up a word and quickly go to the page containing that word. This approach is much quicker than having to check every page for the word you're looking for, or having to go to a specific chapter and a specific section. The direct benefit of using keys is that you don't have to write complex expressions to get to certain nodes. When you use keys, your expressions are likely to be shorter and easier to read. Whether you experience an actual performance benefit when processing is another matter. Some processors might choose to create in internal index while processing, which makes retrieval using keys perform better than using the expression that would be needed if the key didn't exist. Other processors might translate the key to that expression and use it to retrieve the data instead. In that case, it is likely that the performance is not any better, but using keys only makes your life as a programmer easier.

If you're using a specific processor, measuring the performance with and without keys is a good idea. If you measure performance, be sure to use source documents of different sizes. With a small source document, keys might actually be the cause for extra overhead, which doesn't exist for large documents. The result also depends on how much use you make of a key. If you use it only once, using an expression is probably a better idea, because building an internal index takes time as well.

Using Keys to Select Data

You can define a key by using the `xsl:key` element. This is a top-level element, so it might occur only as a child element of the `xsl:stylesheet` element. This makes sense because the key needs to be defined before it ever gets used, so before any processing of elements starts. You can use the `xsl:key` element as follows:

```
<xsl:key name="carkey" match="car" use="@name" />
```

You can see that the `xsl:key` element has three attributes, all of which are mandatory. The `name` attribute gives the key a name, which is necessary because you can define more than one key in a stylesheet. Having multiple keys is like having multiple indexes in a book, such as one with words and one with people, as is sometimes the case in scientific books. The second attribute is `match`, which defines the nodes that need to be indexed. The value of the `match` attribute should be a pattern identifying the nodes that should be indexed. As in the preceding example, the value will be an element name in most cases, but it might also match several elements or attributes. The last attribute is `use`, which tells the processor what it should use to index the matching nodes. In the preceding example, it is an attribute value, but it also can be an expression of some sort. This means that you can use the element's value, the value of one of the child elements, or a combination of values.

After you define a key, you can select elements by using the `key()` function, which returns a node-set with the elements that match the given key. You can use `key()` in an expression by itself or as part of a larger expression. Its use is as follows:

```
key('carkey', 'Focus')
```

The two arguments of the `key()` function are mandatory. The first defines the key you want to use, and the second gives the value you're looking for. Both are string values but can be created using an expression, so you can give dynamic values. Because this is the most common use for the value argument, the samples in this section make use of it. Listing 16.5 shows the source document used for the coming samples.

LISTING 16.5 Sample XML with Cars and Manufacturers

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <car:cars xmlns:car="http://www.example.com/xmlns/car"
3:            xmlns:m="http://www.example.com/xmlns/manufacturer">
4:    <car:models>
5:      <car:model car:name="Golf" m:id="VW" car:year="1999" />
6:      <car:model car:name="Camry" m:id="TY" car:year="1999" />
7:      <car:model car:name="Focus" m:id="FO" car:year="2000" />
8:      <car:model car:name="Civic" m:id="HO" car:year="2000" />
9:      <car:model car:name="Prizm" m:id="CV" car:year="2000" />
10:     <car:model car:name="Celica" m:id="TY" car:year="2000" />
11:     <car:model car:name="Mustang" m:id="FO" car:year="2001" />
12:     <car:model car:name="Passat" m:id="VW" car:year="2001" />
13:     <car:model car:name="Accord" m:id="HO" car:year="2002" />
14:     <car:model car:name="Corvette" m:id="CV" car:year="2002" />
15:   </car:models>
16:   <m:manufacturers>
17:     <m:manufacturer m:id="VW" m:name="Volkswagen" m:country="Germany" />
18:     <m:manufacturer m:id="TY" m:name="Toyota" m:country="Japan" />
19:     <m:manufacturer m:id="FO" m:name="Ford" m:country="USA" />
20:     <m:manufacturer m:id="CV" m:name="Chevrolet" m:country="USA" />
21:     <m:manufacturer m:id="HO" m:name="Honda" m:country="Japan" />
22:   </m:manufacturers>
23: </car:cars>
```

ANALYSIS Listing 16.5 is similar to a listing for yesterday's lesson and familiar for the most part. Line 2 defines the root element, which uses the `car` namespace and declares the namespace. In addition, on line 3, the `m` namespace is declared, denoting manufacturer information. Each `car:model` element uses attributes with a namespace. The same goes for the `m:manufacturer` elements.

In some of the preceding lessons, you learned different ways to combine the car and manufacturer data with data similar to Listing 16.5. Listing 16.6 shows a stylesheet that combines data using a key.

LISTING 16.6 Stylesheet Using a Key

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
4:    xmlns:car="http://www.example.com/xmlns/car"
5:    xmlns:m="http://www.example.com/xmlns/manufacturer">
6:
7:    <xsl:output method="text" encoding="UTF-8" />
8:    <xsl:key name="mfc" match="m:manufacturer" use="@m:id" />
9:
10:   <xsl:template match="/">
11:     <xsl:apply-templates select="/car:cars/car:models" />
12:   </xsl:template>
13:
14:   <xsl:template match="car:models">
15:     <xsl:for-each select="car:model">
16:       <xsl:value-of select="key('mfc', @m:id)/@m:name" />
17:       <xsl:text> </xsl:text>
18:       <xsl:value-of select="@car:name" />
19:       <xsl:text> (</xsl:text>
20:       <xsl:value-of select="@car:year" />
21:       <xsl:text>)&#xA;</xsl:text>
22:     </xsl:for-each>
23:   </xsl:template>
24: </xsl:stylesheet>
```

ANALYSIS On lines 3 and 4, Listing 16.6 declares the namespaces used in Listing 16.5. Line 8 uses these namespaces when it defines a key named `mfc` on the `m:manufacturer` elements. Because this pattern just gives an element name, the elements indexed might occur anywhere in the source document. They don't need to occur at a specific depth or as child elements of some specific element. The key is defined using the `m:id` attribute, which means that you select elements based on the value of the `m:id` attribute. The key can be combined from different nodes, in which case you need to search on the value of the combined nodes. The template on line 10 matching the root element invokes other templates starting at the `car:models` element. This means that the `m:manufacturers` element and its child elements are never processed by a template. The template on line 14 matches the `car:models` element and then uses `xsl:for-each` on line 15 to iterate through all the `car:model` elements. Line 18 outputs the name of the car model, and line 20 outputs the year. The lines in between are just for nice formatting. Line 16 uses the `key()` function to select the `m:manufacturer` element that has an `m:id` attribute with the same value of the `m:id` attribute of the `car:model` element being processed. The `key()`function is part of a larger expression that immediately outputs the name of the manufacturer. Another option would have been to capture the result of the key in a variable and address it separately, but the idea of a key is that you have quick access, so you don't need variables. Listing 16.7 shows the result when this stylesheet is applied to Listing 16.5.

OUTPUT **LISTING 16.7** Result from Applying Listing 16.6 to Listing 16.5

```
Volkswagen Golf (1999)
Toyota Camry (1999)
Ford Focus (2000)
Honda Civic (2000)
Chevrolet Prizm (2000)
Toyota Celica (2000)
Ford Mustang (2001)
Volkswagen Passat (2001)
Honda Accord (2002)
Chevrolet Corvette (2002)
```

ANALYSIS A quick look at Listing 16.7 shows you that each car name is preceded by the manufacturer name. This name was grabbed from another section of the document, similar to another table in a database. Using keys, as such, is a great way to use related data.

From the preceding example, you might think that the value on which a key is based needs to be unique. That is by no means true. If multiple elements have the same key value, the `key()`function will return a node-set with those nodes instead of just one node. You can use that node-set just like any other node-set. Listing 16.8 shows a stylesheet with such a nonunique key.

LISTING 16.8 Stylesheet with Nonunique Key

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" exclude-result-prefixes="car m"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="http://www.example.com/xmlns/car"
  xmlns:m="http://www.example.com/xmlns/manufacturer">

  <xsl:output method="html" encoding="UTF-8" />
  <xsl:key name="cars" match="car:model" use="@m:id" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates select="/car:cars/m:manufacturers" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="m:manufacturers">
    <xsl:for-each select="m:manufacturer">
      <h2><xsl:value-of select="@m:name" /></h2>
```

```
22:       <p><i>Country: <xsl:value-of select="@m:country" /></i></p>
23:       <xsl:for-each select="key('cars', @m:id)">
24:         <ul>
25:           <li>
26:             <xsl:value-of select="@car:name" />
27:             <xsl:text> (</xsl:text>
28:             <xsl:value-of select="@car:year" />
29:             <xsl:text>)</xsl:text>
30:           </li>
31:         </ul>
32:       </xsl:for-each>
33:     </xsl:for-each>
34:   </xsl:template>
35: </xsl:stylesheet>
```

ANALYSIS The first 7 lines of Listing 16.8 are the same as Listing 16.6, except that line 7 defines the output as HTML instead of text. Because the output is HTML, line 2 also tells the processor which namespaces to exclude from the output. Line 8 defines a key on the `car:model` element, using `m:id`. The value of the `m:id` attribute is not unique for each of the `car:model` elements. In fact, for each value of the attribute, there are two `car:model` elements in Listing 16.5. The template on line 10 creates the HTML base code and on line 14 selects the `m:manufacturers` element and invokes the processor to match templates. The template on line 19 matches that element and iterates through all `m:manufacturer` elements in Listing 16.5. Line 23 uses the `key()` function to select any `car:model` elements that have the same value for their `m:id` attribute as the `m:id` attribute of the current `m:manufacturer` element. An `xsl:for-each` element iterates through these elements.

Listing 16.8 is similar to several examples from previous lessons. In fact, the output shown in Listing 16.9 is exactly the same as for those examples. Those examples used the `current()` function or a variable to circumvent the problem that the current node is out of context when the `xsl:for-each` element on line 23 uses a location path. With the `key()` function in this example, that problem doesn't happen, so the value used to select the `car:model` elements can be taken directly from the context node.

OUTPUT **LISTING 16.9** Result from Applying Listing 16.8 to Listing 16.5

```
<html>
   <body>
      <h1>Auto show</h1>
      <h2>Volkswagen</h2>
      <p><i>Country: Germany</i></p>
      <ul>
         <li>Golf (1999)</li>
      </ul>
      <ul>
```

LISTING 16.9 Continued

```
            <li>Passat (2001)</li>
         </ul>
         <h2>Toyota</h2>
         <p><i>Country: Japan</i></p>
         <ul>
            <li>Camry (1999)</li>
         </ul>
         <ul>
            <li>Celica (2000)</li>
         </ul>
         <h2>Ford</h2>
         <p><i>Country: USA</i></p>
         <ul>
            <li>Focus (2000)</li>
         </ul>
         <ul>
            <li>Mustang (2001)</li>
         </ul>
         <h2>Chevrolet</h2>
         <p><i>Country: USA</i></p>
         <ul>
            <li>Prizm (2000)</li>
         </ul>
         <ul>
            <li>Corvette (2002)</li>
         </ul>
         <h2>Honda</h2>
         <p><i>Country: Japan</i></p>
         <ul>
            <li>Civic (2000)</li>
         </ul>
         <ul>
            <li>Accord (2002)</li>
         </ul>
      </body>
   </html>
```

One of the great things about keys is that you can define a key for different elements. If you use the `key()` function with such a key, any element that matches is returned. Listing 16.10 shows an XML sample that can benefit from this functionality.

LISTING 16.10 Sample XML with Movies

```
<?xml version="1.0" encoding="UTF-8"?>
<movies>
  <movie title="The Good, the Bad, and the Ugly" genre="western">
    <director name="Sergio Leone" />
    <actor name="Clint Eastwood" character="Biondo" />
    <actor name="Lee Van Cleef" character="Angel Eyes Sentenza" />
    <actor name="Eli Wallach" character="Tuco Ramirez" />
  </movie>
  <movie title="The Piano" genre="drama">
    <director name="Jane Campion" />
    <actor name="Holly Hunter" character="Ada McGrath" />
    <actor name="Harvey Keitel" character="George Baines" />
  </movie>
  <movie title="Bird" genre="drama">
    <director name="Clint Eastwood" />
    <actor name="Forest Whitaker" character="Charlie 'Bird' Parker" />
  </movie>
  <movie title="Stir Crazy" genre="comedy">
    <director name="Sidney Poitier" />
    <actor name="Gene Wilder" character="Skip Donahue" />
    <actor name="Richard Pryor" character=" Harry Monroe" />
  </movie>
</movies>
```

ANALYSIS Listing 16.10 contains several `movie` elements with information about a movie. Each `movie` element has a `title` and `genre` attribute and several `actor` and `director` child elements. These elements have at least a `name` attribute with the name of the actor or director. Listing 16.11 shows a stylesheet that selects data with a key matching multiple elements.

LISTING 16.11 Stylesheet with Key Matching Different Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:key name="searchkey" match="actor" use="@name" />
  <xsl:key name="searchkey" match="director" use="@name" />
  <xsl:param name="search" />

  <xsl:template match="/">
    <xsl:for-each select="key('searchkey',$search)">
      <xsl:value-of select="@name" />
      <xsl:text>&#xA;-</xsl:text>
      <xsl:value-of select="name()" />
      <xsl:text>: </xsl:text>
      <xsl:value-of select="../@title" />
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS On lines 6 and 7 in Listing 16.11, the `xsl:key` element is used twice for the same key name. The `match` attribute for the two elements is different. The effect is that the `searchkey` key applies to both the `actor` and `director` elements. A parameter defined on line 8 is later used as the value to search for in the key. On line 11, the `key()` function selects `actor` and `director` elements that have a `name` attribute identical to the value of the `search` parameter. The body of the `xsl:for-each` element on line 11 outputs that name on line 12 and whether it is an actor or director on line 14. Line 16 goes up a level to the `movie` element and outputs the title of the movie. The rest of the body is just there for formatting. When Listing 16.11 is applied to Listing 16.10 and the parameter passed has the value `Clint Eastwood`, Listing 16.12 is the output.

OUTPUT **LISTING 16.12** Result from Applying Listing 16.11 to Listing 16.10

```
Clint Eastwood
-actor: The Good, the Bad, and the Ugly
Clint Eastwood
-director: Bird
```

ANALYSIS The `key()` function in Listing 16.11 returns Clint Eastwood as actor in one movie and director of another. This was the whole idea of using the combined key. Each element is used with its surrounding elements to get the output in Listing 16.12.

In Listing 16.11, the key is defined with two `xsl:key` elements. Because the `use` attribute has the same value for both, you can replace those two lines with one `xsl:key` element, which would look like this:

```
<xsl:key name="searchkey" match="actor|director" use="@name" />
```

The `match` attribute in the preceding sample uses a union pattern to combine two elements for the key. Such a pattern can be more complex, of course. You need two separate `xsl:key` elements if the `use` attribute must have a different value. This would be the case if you were searching for movies and the `director` element was an attribute of the `movie` element. In that case, the key should be defined as follows:

```
<xsl:key name="searchkey" match="movie" use="actor/@name" />
<xsl:key name="searchkey" match="movie" use="@director" />
```

Caution You can't define a key dynamically using variables and parameters.

Working with Unique IDs

Document Type Definitions (DTDs) enable you to define an attribute of an element as an ID attribute. This type of attribute has to be unique for that element across the entire XML document, so the attribute is a unique identifier for that element. This property is useful to track down elements in a way similar to when you use keys. Keys are much more flexible, but if you work with documents that already use a DTD that defines an ID attribute, why not use it? Listing 16.13 shows an XML document with an internal DTD.

Note

Discussing DTDs in detail is beyond the scope of this book. Apart from ID attributes, DTDs are not likely to have any effect on how you create a stylesheet.

LISTING 16.13 XML Document with an Internal DTD

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <!DOCTYPE manufacturers [
3:    <!ELEMENT manufacturers (manufacturer)*>
4:    <!ELEMENT manufacturer EMPTY>
5:    <!ATTLIST manufacturer
6:      id      ID     #REQUIRED
7:      name    CDATA  #REQUIRED
8:      country CDATA  #REQUIRED>
9:  ]>
10: <manufacturers>
11:   <manufacturer id="VW" name="Volkswagen" country="Germany" />
12:   <manufacturer id="TY" name="Toyota" country="Japan" />
13:   <manufacturer id="FO" name="Ford" country="USA" />
14:   <manufacturer id="CV" name="Chevrolet" country="USA" />
15:   <manufacturer id="HO" name="Honda" country="Japan" />
16: </manufacturers>
```

ANALYSIS Lines 2 through 9 in Listing 16.13 define an internal DTD that is used to validate the XML when it is loaded by a validating parser. Line 3 tells the parser that the document might have a `manufacturers` element that can contain zero or more `manufacturer` elements. Line 4 defines the `manufacturer` element as being empty, meaning that it cannot have any content, except for attributes. Lines 5 through 8 define the attributes for the `manufacturer` attribute. The definitions correspond to the attributes that these elements have in the XML document. For each attribute, the data type is defined. In the case of the `name` and `country` attributes, this is character data (string), which is denoted by `CDATA`. The definition of the `id` attribute on line 6 tells the parser that this attribute is of type `ID`, which means that the parser has to make sure that the values of this attribute are unique for each element throughout the document. Each attribute is defined as `REQUIRED`, which means that the element must have this attribute, or it is invalid.

In the preceding example, the attribute that is defined of type `ID` is named `id`. It is a misconception that an `ID` type attribute must have the name `id` or `ID`. The requirement is that the values are unique; you can name the attribute any way you want. The following defines an attribute named `ssn` as an `ID` attribute:

```
<!ATTLIST Person
  ssn ID #REQUIRED>
```

Selecting Data with a Unique ID

An element that has an `ID` type attribute can be selected using the `id()` function. This function resembles the `key()` function, but it has only one argument—the value of the attribute you're looking for. The `id()` function returns the corresponding element for which the `ID` type attribute is defined. Listing 16.14 shows this function in action.

LISTING 16.14 Stylesheet Selecting on a Unique ID

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:template match="/">
8:      <xsl:value-of select="id('FO')/@name" />
9:    </xsl:template>
10: </xsl:stylesheet>
```

ANALYSIS Listing 16.14 is simple. It has only one template, and on line 8, it outputs one value—the value of the `name` attribute of the `manufacturer` element that has an `id` attribute with the value `FO`. Listing 16.15 shows the result when Listing 16.14 is applied to Listing 16.13.

OUTPUT **LISTING 16.15** Result from Applying Listing 16.14 to Listing 16.13

```
Ford
```

ANALYSIS Listing 16.15 is not much to look at because there is only one element in Listing 16.13 for which the `id` attribute has the value `FO`. This element has a `name` attribute with the value `Ford`. So, you can see that the `id()` function returns the node corresponding to the given `ID` value.

IDs, IDREFS, and XML Schema

Closely related to the `ID` type attributes in a DTD are IDREFS, which are references to `ID` type attributes defining a relationship between different elements. XSLT supports only `ID` type attributes and has no support for IDREFS.

XML Schema uses a different mechanism entirely for this functionality. This mechanism closely resembles keys in XSLT. XML Schema is not yet supported by XSLT, so unfortunately it isn't of much use to XSLT developers at this time. XML Schema support will not be available in XSLT before version 2.0.

Inserting Unique IDs

Suppose you want to create an XML document that will be validated by an external DTD. In that case, you need to have some way of making sure that the attributes that are designated as `ID` attributes in the DTD contain a unique value. You can achieve this by creating the values for these attributes with the `generate-id()` function, which returns a string with a unique identifier. You can use this function without an argument, in which case it will generate a unique identifier based on the context node. You also can pass a parameter that points to a certain node using a pattern. The key is then generated based on that node. If the argument is a node-set rather than a node, the result is based on the first node in the node-set. The other nodes are ignored. The idea here is that, for each node in the document, the `generate-id()` function returns a unique value. If you use the `generate-id()` function later in a document for the same node, the result is the same. This feature is useful when you create a document in which elements reference each other—for instance, when you create an HTML document with links to specific places in a document. Because this approach is actually much more interesting than creating an XML document with an `ID` type attribute, the next example will focus on it. In the exercise at the end of this lesson, you will create an XML document with unique identifiers. Listing 16.16 shows a stylesheet that creates an HTML document with referring links based on the generated `ID` value of a node.

LISTING 16.16 Stylesheet Generating IDs Linking to Other Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" exclude-result-prefixes="car m"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="http://www.example.com/xmlns/car"
  xmlns:m="http://www.example.com/xmlns/manufacturer">

  <xsl:output method="html" encoding="UTF-8" />
  <xsl:strip-space elements="*" />
  <xsl:key name="mfc" match="m:manufacturer" use="@m:id" />

```

LISTING 16.16 Continued

```
11:   <xsl:template match="/">
12:     <html>
13:     <body>
14:       <xsl:apply-templates select="/car:cars/car:models" />
15:       <hr />
16:       <xsl:apply-templates select="/car:cars/m:manufacturers" />
17:     </body>
18:     </html>
19:   </xsl:template>
20:
21:   <xsl:template match="car:models">
22:     <h2>Car Models</h2>
23:     <ul>
24:       <xsl:for-each select="car:model">
25:         <li>
26:           <a href="#{generate-id(key('mfc', @m:id))}">
27:             <xsl:value-of select="key('mfc', @m:id)/@m:name" />
28:           </a>
29:           <xsl:text> </xsl:text>
30:           <xsl:value-of select="@car:name" />
31:           <xsl:value-of select="concat(' (',@car:year,')')" />
32:         </li>
33:       </xsl:for-each>
34:     </ul>
35:   </xsl:template>
36:
37:   <xsl:template match="m:manufacturers">
38:     <h2>Manufacturers</h2>
39:     <ul>
40:       <xsl:apply-templates />
41:     </ul>
42:   </xsl:template>
43:
44:   <xsl:template match="m:manufacturer">
45:     <li>
46:       <a name="{generate-id()}">
47:         <xsl:value-of select="@m:name" />
48:       </a>
49:         <xsl:value-of select="concat(' (',@m:country,')')" />
50:     </li>
51:   </xsl:template>
52: </xsl:stylesheet>
```

ANALYSIS Listing 16.16 uses a mix of techniques. Because this stylesheet is used with Listing 16.5, it needs to declare the same namespaces, which it does on lines 4 and 5. Line 9 defines a key on the `m:manufacturer` elements for quick access. The root template on line 11 creates the HTML base code and uses `xsl:apply-templates` twice:

on line 14 for the cars and on line 16 for the manufacturers. The former matches the template on line 21, which creates a header and iterates through the `car:model` elements with line 24. Line 26 creates a link using the `generate-id()` function. The `key()` function provides that function with the element the ID is generated for, which is the `m:manufacturer` element belonging to the `car:model` element being processed. The same key is used on line 27 to get the name of the manufacturer. The rest of that template outputs the values of the current `car:model` element plus some formatting. The manufacturers are processed by two templates: The template on line 37 just takes care of the outer shell, and the template on line 44 actually outputs the values and the formatting for each manufacturer. Line 46 creates an HTML anchor that the link discussed earlier links to. To accomplish this, the anchor needs to get the same ID value. The `generate-id()` function creates an ID for the context node, which is an `m:manufacturer` node, so the ID value is, in fact, the same as that created on line 26. The rest of the template is again mostly values and formatting. Listing 16.17 shows the result.

OUTPUT **LISTING 16.17** Result from Applying Listing 16.16 to Listing 16.5

```
<html>
   <body>
      <h2>Car Models</h2>
      <ul>
         <li><a href="#d1e14">Volkswagen</a> Golf (1999)
         </li>
         <li><a href="#d1e15">Toyota</a> Camry (1999)
         </li>
         <li><a href="#d1e16">Ford</a> Focus (2000)
         </li>
         <li><a href="#d1e18">Honda</a> Civic (2000)
         </li>
         <li><a href="#d1e17">Chevrolet</a> Prizm (2000)
         </li>
         <li><a href="#d1e15">Toyota</a> Celica (2000)
         </li>
         <li><a href="#d1e16">Ford</a> Mustang (2001)
         </li>
         <li><a href="#d1e14">Volkswagen</a> Passat (2001)
         </li>
         <li><a href="#d1e18">Honda</a> Accord (2002)
         </li>
         <li><a href="#d1e17">Chevrolet</a> Corvette (2002)
         </li>
      </ul>
      <hr>
      <h2>Manufacturers</h2>
      <ul>
         <li><a name="d1e14">Volkswagen</a> (Germany)
         </li>
```

LISTING 16.17 Continued

```
31:          <li><a name="d1e15">Toyota</a> (Japan)
32:          </li>
33:          <li><a name="d1e16">Ford</a> (USA)
34:          </li>
35:          <li><a name="d1e17">Chevrolet</a> (USA)
36:          </li>
37:          <li><a name="d1e18">Honda</a> (Japan)
38:          </li>
39:       </ul>
40:    </body>
41: </html>
```

ANALYSIS If all is well, Listing 16.17 should contain links from each car to its manufacturer. The links are based on ID values created with the `generate-id()` function. So, for the cars from the same manufacturer, the ID value should be the same and link to the manufacturer with that ID. For the Volkswagen cars, this means that the value of the `href` attributes on lines 5 and 19 should match the value of the `name` attribute on line 29. As you can see, that is actually the case. If you check the other cars and manufacturers, you will find that the value is correct for each of them, as it should be.

Using Keys and Generated IDs to Select Distinct Values

Earlier in this lesson, you learned how to select only the distinct values in a node-set with values that might have duplicate values. You learned that you can use the `preceding-sibling` axis to accomplish this task, but that with large node-sets, this might not perform well. Providing a processor uses an internal index when creating a key, keys can be used to better the performance with large node-sets. Listing 16.18 shows you how this works.

LISTING 16.18 Stylesheet Selecting Distinct Values Based on a Key

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:key name="distinct" match="model" use="@manufacturer" />
8:
9:    <xsl:template match="/">
10:     <xsl:for-each select="//model[generate-id() =
11:          generate-id(key('distinct', @manufacturer)[1])]/@manufacturer">
12:       <xsl:sort select="." />
```

```
13:       <xsl:value-of select="concat(.,'&#xA;')" />
14:     </xsl:for-each>
15:   </xsl:template>
16: </xsl:stylesheet>
```

ANALYSIS Listing 16.18 is the same as Listing 16.3 for the most part. Line 7 defines a key for all the `model` elements. The key is generated based on the value of the `manufacturer` attribute, which in Listing 16.2 is not unique for each `model` element. Line 10 contains the expression selecting the distinct values, which is totally different from that used on lines 8 and 9 in Listing 16.3. The predicate that makes sure you get distinct values uses the `key()` and `generate-id()` functions to filter out the duplicate nodes. The expression is based on the fact that the `generate-id()` function always creates the same ID value for the same node. The `key()` function selects all the `model` elements that have the same value for the `manufacturer` attribute. Only the first of those should be sent to the output, which means that the generated ID value of the current `model` element should be the same as that of the first node in the node-set returned by the `key()` function. Listing 16.18 will yield the exact same result as Listing 16.3. Listing 16.18 is likely to perform better with large node-sets.

There is another alternative for the predicate expression in Listing 16.18. If you select the current node and the first node returned by the `key()` function, they should be the same. This means that a union of these two elements would actually be one element, so when you count the number of nodes in the node-set, the result should be 1. This is expressed by the following predicate:

```
count(.|key('distinct', @manufacturer)[1]) = 1
```

This expression counts the number of nodes in the union (expressed by the pipe symbol) of the current node and the first in the node-set returned by the `key()` function.

Summary

Today you learned about some intricate features of expressions. These features can give you a hard time because they can be the cause for results other than you expected. These problems are related to implicit type conversion 99 out of 100 times. It is imperative that you learn these rules well, so rereading the lesson from Day 10 wouldn't be a bad idea.

You also learned that you can use keys to retrieve data easily, in the process probably increasing the performance of your stylesheet. Keys allow you a quick access method to those elements for which you have defined a key. You can use nodes of the same name and type for the key, but you also can use nodes of different types and names. In addition, these nodes can be scattered throughout a document, depending on the `match` expression you use to create the index.

A last method to quickly retrieve elements is tied into the use of `ID` type attributes defined in a DTD. These attributes must have a unique value across a document, so they can be used by the processor to retrieve elements based on that value, no matter where the location of the element. Because such an attribute needs to be defined in a DTD, this functionality is useful only in valid XML documents that have `ID` type attributes defined.

In tomorrow's lesson, you'll learn about recursion, which is, among other things, important for computational stylesheets. Recursion is an important mechanism to get around some of the problems with variables.

Q&A

Q Is there a limit to the number of keys I can create?

A No. You can use as many keys as you want. You should, however, use keys only if it really makes sense from a performance point of view or if using them greatly simplifies your expressions. I suggest testing the key performance of your processor of choice before committing to an implementation using keys if performance is critical.

Q Can I search for more than one ID value at a time?

A Yes. The argument of the `id()` function can actually be a whitespace-separated list of ID values searched for. All elements that match are returned as a node-set.

Q Why are IDREFS not supported in XSLT?

A Unless you have access to the DTD, IDREFS don't really make much sense in XSLT because you can't know which values are actually IDREFS. If you do know, you know at design time, so you can use that information when creating the stylesheet.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Values used as a key need to be unique.
2. True or False: The `generate-id()` function always creates the same value if used on the same node in a document.

3. If you compare a string or number value to a node-set, when will the comparison say that the values are equal?
4. Can you benefit from storing the result of the `key()` function in a variable?
5. Why can the value of an `ID` type attribute be used to retrieve data quickly?

Exercise

1. Create a stylesheet for Listing 16.13. This stylesheet should re-create this document (excluding the DTD), but with the values of `id` attribute created with `generate-id()`. The original value might be discarded or put in another attribute named `code`.

This page intentionally left blank

DAY 17

Using Recursion

In yesterday's lesson, you learned some additional techniques for selecting and matching data in an XML source. You learned that keys and ID type values defined in a Document Type Definition (DTD) can speed up data retrieval, or at the very least make it easier to select certain data.

NEW TERM Today's lesson is again about operating on data and will cover only one important topic: *recursion*. Recursion is a technique rather than an element or a function, and it is mostly used in situations in which iteration isn't sufficiently powerful enough.

In today's lesson, you will learn the following:

- What recursion is
- How to use recursion
- How recursion can help you get around some of XSLT's limitations
- What the drawbacks of recursion are

Understanding Recursion

Recursion is a powerful, useful, but sometimes tricky technique that can be used in place of iteration, which you're already familiar with. Recursion and iteration are similar, but with recursion you can perform tasks that are not possible with simple iteration.

What Is Recursion?

Recursion is a common construct in programming. It occurs when a function or a template calls itself. A function calling itself sounds a little odd because a function is called to perform a certain task. Why call it again to perform that task? The idea is that the task the function needs to perform can be broken into smaller tasks. These tasks are the same as the original task, but smaller. To solve that task, you can call the same function, which will, in turn, break the task into smaller pieces and call itself again. This process goes on until the task is so simple that it can be performed without calling the function again. The last time the function is called, it returns a result. The function that called the last function performs its small part of the work and tags on the result from the function it called. It then returns the result so that the function that called it can do the same. This process goes on until the first function that was called does its part and tags on the result from the function it called. It then returns the final result to its caller.

Your head is probably spinning by now, so let's look at a common example of recursion in mathematics. The factorial of 5, written as `5!`, is a great example of recursion. `5!` is equal to `5*4*3*2*1`. The result can be calculated by breaking the calculation into several smaller pieces—for instance, `(5*4)*(3*2)*1`, which is equal to `20*6*1`. If you calculate the result of `5!` using recursions, it looks like the representation shown in Figure 17.1.

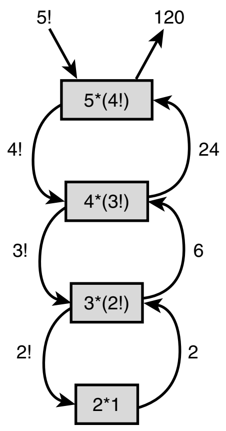


FIGURE 17.1
Visual representation of recursive function calculating `5!`.

The boxes in Figure 17.1 represent the function that is called to perform the calculation. The arrows on the left show the call with the value given to calculate. In the boxes, you can see how the function breaks down the calculation. Each time, the function takes off one multiplication and then calls itself to calculate the remainder, which is shown between the parentheses. The arrows on the right show the result of each step in the calculation. You can check that the result of `5!` is indeed 120 by substituting the factorial in the box with the result calculated up until then. The smallest box no longer has to break down the calculation because only two numbers are left to multiply. The result of that multiplication is multiplied by the number that was taken off by the function above it. This process continues until the top function returns the final result. You can see that for each calculation the same algorithm is reused to create the desired result.

Why and When Should You Use Recursion?

Using recursion is an elegant way to solve problems. If you can define a problem in its smallest form and create a function or template to solve that problem, that problem can be solved no matter how big it is. The only limit is the one imposed by the computer and/or the processor. Because of the way a recursive function solves a problem, it is often quite small—smaller than any code that solves the same problem without recursion. In XSLT, several tasks are impossible to perform without recursion; the calculation shown in Figure 17.1 is one of them.

So, why can't you solve such problems without recursion? The answer is quite simple: The value of a variable is unchangeable as long as it is in context. Therefore, you can't use a simple calculation line such as `myvar = myvar + 1`. If that weren't the case, you could have used iteration instead of recursion to solve the factorial example. For instance, in Visual Basic, you would make the same calculation as follows:

```
Result = 1
For i = 1 To Factorial
  Result = Result * i
Next
```

This code is much simpler because, with each iteration, the value of `i` is increased by 1, and the result is created from what you already had. When you use recursion, this use of a variable is actually simulated each time you pass the parameter's new value to the next template call.

Why Can’t Variables in XSLT Change?

Variables in XSLT can’t change for a reason. At the core of this reason is the fact that a variable that can change implies an order in which commands should be executed. If the order is different, the value of the variable is different when certain commands are executed, so the result would be different. This undermines the idea that a function or a template should always return the same result if the input is the same, which is a central principle in XSLT. This principle is rooted in mathematics, where an expression such as `X + X` should always yield the same result if `X` is the same. So, if `X` equals 5, the result should always be 10, no matter when you use the expression or how many times you use it. If an expression is based on a changing value, this is not possible.

The fact that functions and expressions always yield the same result means that there are no side effects when executing XSLT code. This means that a processor could do tasks in parallel—for instance, processing several nodes in a node-set in parallel—because it doesn’t matter when a task is performed. The processor can aggregate the results and put them in the order that should be sent to the output.

In theory, it is also possible to perform operations incrementally. This means that a processor can start processing a document that hasn’t fully been (down)loaded yet. In a distributed environment such as the Internet, this has the benefit that you can start to show the user something before a page is fully transferred to the client. In addition, this means that you can have a base result that can be changed later, such as a map with a weather forecast. Instead of reloading the entire image, the map can be reused and just the forecast information changed.

Both parallel and incremental processing with XSLT has still to be implemented. Because of the enormous benefits, it is only a matter of time until some of the major vendors will have processors that can do either or both.

A textbook example of a problem that can be solved well with recursion involves a traveling salesman. In this problem, the salesman has to visit a number of cities. You have to find the quickest way for the salesman to visit all of them. The way to solve this problem is to break the tasks into smaller pieces again. Say the salesman has to visit five cities. In that case, you start with a city and then calculate the quickest way to visit the other four cities. You do so by calculating the quickest way to visit the remaining three, plus the time for the fourth city. Just as with the factorial problem, you end up with just the time required to go from one city to another, which is easy to calculate.

Such problems occur most of the time in XSLT when there is no way to iterate using templates or `xsl:for-each`. This situation happens most often when you have to operate on the value of an attribute or element with no child elements or anything. In that case, you have just the raw data. Consider the following example:

```
<name>Tuco Benedito Pacifico Juan Maria Ramirez</name>
```

You're probably thinking that it would be better to store this name as follows (and you would be right):

```
<name>
  <firstname>Tuco</firstname>
  <firstname>Benedito</firstname>
  <firstname>Pacifico</firstname>
  <firstname>Juan</firstname>
  <firstname>Maria</firstname>
  <lastname>Ramirez</lastname>
</name>
```

The second format is much better because you can work with the first names separately and separate from the last name. If you wanted to output Tuco B. P. J. M. Ramirez, you could easily do so with the second format. You can just use `xsl:for-each` to iterate through each `firstname` element and then use the `substring()` function to get the first letter. With the first format, you can't do that. You have to keep track of where you are in the string somehow. The only solution here is to use recursion. This approach would, of course, also enable you to reformat the first format into the second. If you designed the original document, you probably would go for the second format, but you will find that when you start working with XML documents created by others, data such as that used in the first format is not uncommon. The only thing to do is to deal with it. Recursion gives you the opportunity to do so.

The Drawbacks of Recursion

The fact that recursion is elegant and can solve problems that are otherwise unsolvable doesn't mean you should always use it. When you can solve the same problem by using template matching or `xsl:for-each`, go for it. In most cases, such a solution will perform better. Depending on how smart the processor is (or actually the compiler that's part of the processor), recursion can cost a lot of memory and performance. When you use recursion, the processor has to keep track of how many times a template has been called and what the state of each template is. Doing so costs memory, and this memory usage can increase rapidly. When the processor runs out of memory, it will fail, and fail badly. The likelihood that this failure will happen with matching or iteration is much smaller because the memory usage of these mechanisms is much lower.

New Term Whether recursion performs poorly also depends on the compiler. If the compiler can detect that a function is recursive, it can cut down on both memory usage and execution time because it can treat the recursive code almost as if it is iterative code. Detecting whether code is recursive, however, is by no means easy. A variant of recursion that is relatively easy to track down is called *tail recursion*, which is the case when a template calls itself on the last line of the template (or at least if there are only closing tags after it).

A major problem of recursion is that it needs to end somewhere; otherwise, the template would just keep calling itself until the program fails. So, you need to make sure that the template checks when it has to return to its caller without calling itself again. With a simple template, this check is easy, and either you return immediately, or you call the same template again. When this check becomes complex, the likelihood increases that you haven't covered all the bases, in which case you get a never-ending piece of code—never ending, that is, until the processor fails.

Creating Recursive Templates

In XSLT, you can't create recursive functions. In fact, you can't create functions. Period. The only way you can create functions is to extend the processor, which is the topic of Day 20, and cannot be considered as XSLT itself. That said, you can create recursive templates.

Recursive templates call themselves using `xsl:call-template`. Using `xsl:apply-templates` wouldn't make sense because then you have no way of knowing for sure that the correct template is processed. Recursion builds on the fact that it processes the correct template because each time the template is called, a smaller version of the original task has to be executed. If you go on to an entirely different template all of a sudden, then that is no longer true. One exception here occurs when you have templates that call each other in a circular fashion; for instance, template A calls template B, which calls template C, which calls template A again. Although, in the strictest sense, this example is not recursion, it is the same thing; the functionality is just broken into several templates instead of everything in one.

The basis of a recursive template looks like this:

```
<xsl:template name="recursivetemplate">
  <xsl:call-template name="recursivetemplate" />
</xsl:template>
```

The preceding code is not enough. The template will call itself again and again. There is never a way out, so eventually this code will fail because the processor will run out of memory. To solve this problem, you should create the code so that it at least looks like this:

```
<xsl:template name="recursivetemplate">
  <xsl:if test="recursionclause">
    <xsl:call-template name="recursivetemplate" />
  </xsl:if>
</xsl:template>
```

The `xsl:if` element makes sure that the template calls itself only if *`recursionclause`* returns `true`. This is actually the tricky bit because the result of this clause will not change unless you have some way of changing values. Variables in XSLT, however, cannot change, and even if they could, they go out of scope. The solution is to use parameters, so you can create a template that returns a result based on the parameter. Because the template also has to return values, more changes over the preceding code are required. Listing 17.1 shows a stylesheet with a proper recursive template that actually implements the factorial function discussed in the preceding section.

LISTING 17.1 Stylesheet Implementing the Factorial Function

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:call-template name="calculate">
      <xsl:with-param name="num">6</xsl:with-param>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="calculate">
    <xsl:param name="num">1</xsl:param>
    <xsl:choose>
      <xsl:when test="$num = 1">
        <xsl:value-of select="$num" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="result">
          <xsl:call-template name="calculate">
            <xsl:with-param name="num" select="$num - 1" />
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$result * $num" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

Analysis The recursive template in Listing 17.1 starts on line 13. On line 14, a parameter named `num` is defined with a default value of 1. Whether the template calls itself again is decided with an `xsl:choose` element. The only `xsl:when` element is on line 16. It checks whether the value of the `num` parameter is 1. If it is, the last value of the factorial multiplication row is reached, and no multiplication is needed. Therefore, only the value of the `num` parameter has to be returned, which is done on line 17. Any other code is ignored, so the call for which the test expression on line 16 is `true` is the last template to be called—the escape hatch, as it were.

What happens inside the `xsl:otherwise` element is actually much more interesting. The template calling the same template again on line 21 sits inside a variable, defined on line 20. So, any values written in the called template will become the value of the variable. That result is the result of the same task performed on a smaller number. This is what the `xsl:with-param` element on line 22 is for. It calls the same template again, but with the value of the `num` parameter minus 1. The result of this call is multiplied by the current value of the `num` parameter. Therefore, the result returned to the caller is the result from the called template multiplied by the value of the `num` parameter within the current template. This stylesheet doesn't actually depend on any XML source. Line 9 calls the `calculate` template with the hard-coded value 6. So, if all is well, the result is always 720. Listing 17.2 is much more intelligent.

Note

You can run Listing 17.1 with MSXSL, Saxon, or Xalan from the command line. Any XML file (including the stylesheet itself) is valid input, because line 9 has a hard-coded value, so the stylesheet doesn't require a source document.

Listing 17.2 Stylesheet Showing Recursive Steps

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:param name="factorial">1</xsl:param>

  <xsl:template match="/">
     <xsl:call-template name="calculate">
      <xsl:with-param name="num" select="$factorial" />
    </xsl:call-template>
  </xsl:template>

```

```
15:   <xsl:template name="calculate">
16:     <xsl:param name="num">1</xsl:param>
17:     <xsl:message>param: <xsl:value-of select="$num" /></xsl:message>
18:     <xsl:choose>
19:       <xsl:when test="$num = 1">
20:         <xsl:value-of select="$num" />
21:             <xsl:message>return value: <xsl:value-of select="$num" />
                ➥</xsl:message>
22:       </xsl:when>
23:       <xsl:otherwise>
24:         <xsl:variable name="result">
25:           <xsl:call-template name="calculate">
26:             <xsl:with-param name="num" select="$num - 1" />
27:           </xsl:call-template>
28:         </xsl:variable>
29:         <xsl:value-of select="$result * $num" />
30:         <xsl:message>return value: <xsl:value-of
            ➥ select="$result * $num" /></xsl:message>
31:       </xsl:otherwise>
32:     </xsl:choose>
33:   </xsl:template>
34: </xsl:stylesheet>
```

ANALYSIS Listing 17.2 depends on a global parameter that can be set from outside the stylesheet. Setting this parameter enables you to run the code for different numbers. In addition, lines 17, 21, and 30 contain an `xsl:message` element. This element doesn't create any visible output normally but writes a message to a separate output stream. This output stream is used for reporting, such as reporting errors or timing information. When you run a processor from the command line, this output also appears, but it doesn't interfere with processing or alter the output from the stylesheet itself. You can see this very well when you tell the processor to send the output to a file. For instance, with Saxon, you can type

```
saxon -o out.txt 17list02.xsl 17list02.xsl factorial=7
```

With MSXSL, you can run the stylesheet from the command line as follows:

```
msxsl -o out.txt 17list02.xsl 17list02.xsl factorial=7
```

Caution

Although the MSXML parser/processor component supports the `xsl:message` element, the MSXSL command-line tool doesn't send the messages to the output. They can be accessed only when the component is used from code in an application.

This command sends the output from the transformation to the file named `out.txt`. Note that because you don't need an XML source, you can give any XML source. I just used the stylesheet as XML source as well. The output in the file will just be the result, which is `5040`. In the command window, you can see the output from the `xsl:message` elements. You can use `xsl:message` during development to keep track of what's happening, similar to `xsl:comment`, with the difference that `xsl:comment` creates output as part of the stylesheet's result. This means that `xsl:comment` is not usable in a situation in which a template returns a result that should be processed.

Recursion with Single Values

As I explained in the preceding sections, working with single data values that arguably should have been broken up into elements or attributes can cause trouble. When such a value has a rigid structure, you can get away without using recursion because you know the structure at design time. This means you can break up the value into discrete pieces with functions such as `substring()`, `substring-after()`, and `substring-before()`. Data like that shown in Listing 17.3 is an entirely different matter.

LISTING 17.3 XML

```
<?xml version="1.0" encoding="UTF-8"?>
<names>
  <name>Tuco Benedito Pacifico Juan Maria Ramirez</name>
  <name>John Fitzgerald Kennedy</name>
</names>
```

ANALYSIS The data values in Listing 17.3 are dynamic. By that, I mean that you have no idea at design time how many first names a full name contains. Therefore, you can't just use `substring-after()` and `substring-before()` to get to each of the first names. Well, you can, but not without recursion. Recursion enables you to operate on such data because it enables you to dynamically move through the data. Listing 17.4 shows this use effectively.

LISTING 17.4 Stylesheet Using Recursion to Initial First Names

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:apply-templates select="names/name" />
  </xsl:template>

```

```
11:   <xsl:template match="name">
12:     <xsl:call-template name="initial">
13:       <xsl:with-param name="fullname" select="normalize-space()" />
14:     </xsl:call-template>
15:     <xsl:text>&#xA;</xsl:text>
16:   </xsl:template>
17:
18:   <xsl:template name="initial">
19:     <xsl:param name="fullname" />
20:     <xsl:choose>
21:       <xsl:when test="contains($fullname,' ')">
22:         <xsl:value-of select="substring($fullname,1,1)" />
23:         <xsl:text>. </xsl:text>
24:         <xsl:call-template name="initial">
25:           <xsl:with-param name="fullname"
26:                select="substring-after($fullname,' ')" />
27:         </xsl:call-template>
28:       </xsl:when>
29:       <xsl:otherwise>
30:         <xsl:value-of select="$fullname" />
31:       </xsl:otherwise>
32:     </xsl:choose>
33:   </xsl:template>
34: </xsl:stylesheet>
```

ANALYSIS Listing 17.4 initials all the first names of the full names in Listing 17.3. This process is mostly performed by the recursive template on line 18. The value it processes is the `fullname` parameter defined on line 19. The template assumes that the string value it gets is normalized, so before the template is called for the first time, line 13 normalizes the name value. This means that all names are separated only by spaces and that there is no leading or trailing whitespace. The escape hatch for the recursive template is the situation in which no more spaces exist in the name passed to the template. At that point, only the last name is left. The `xsl:otherwise` element on line 29 is there to deal with this situation. All that is done in this case is writing the name, which occurs on line 30. The recursive part of the template is handled by the `xsl:when` element on line 21; this part is processed if spaces still exist in the name—in other words, when first names are still left. Because first names still exist, the first letter of the entire value is the first letter of the first name to be initialed. Line 22 therefore uses the `substring()` function to write just the first letter of the value. Line 23 follows up with a period and space. Line 24 calls the template again, with line 25 providing the new name to be processed. This name should be the current value minus the first name just handled. Because the names are separated by spaces, you can easily get this value by using the `substring-after()` function. Applying Listing 17.4 to Listing 17.3 yields the result in Listing 17.5.

OUTPUT **LISTING 17.5** Result from Applying Listing 17.4 to Listing 17.3

```
T. B. P. J. M. Ramirez
J. F. Kennedy
```

Totaling with Recursion

The computational capabilities of XSLT are limited. In fact, they are so limited that for computations that go just a little beyond the basics, you have to be creative. In many cases, you end up with a recursive solution. This section discusses a common scenario for e-commerce sites, where you have an inventory and an order combined with the inventory data to create an invoice, and so on. The example given here is based on the familiar menu document shown in Listing 17.6.

LISTING 17.6 XML Source Representing an "Inventory"

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

ANALYSIS In the menu data in Listing 17.6, each `dish` element has an `id` attribute. In the preceding chapters, you didn't use this attribute. Now, however, these values are needed to uniquely identify each element because the order information in Listing 17.7 refers to these `id` attributes to define the order.

LISTING 17.7 XML Source with Order Data Related to Listing 17.6

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order id="2" quantity="1" />
  <order id="3" quantity="2" />
  <order id="4" quantity="2" />
  <order id="5" quantity="3" />
  <order id="7" quantity="2" />
  <order id="10" quantity="1" />
  <order id="12" quantity="4" />
</orders>
```

ANALYSIS Listing 17.7 shows an order from a party of five taken by the waiter (probably after he punched it into the computer). Each dish ordered is identified by the `id` attribute; how many times that dish is ordered is given by the `quantity` attribute. When the dinner party asks for the check, Listing 17.8 springs into action and creates a nice-looking list of dishes ordered and the total.

LISTING 17.8 Stylesheet That Creates a Check Based on Listings 17.6 and 17.7

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />
  <xsl:variable name="dishes" select="document('17list06.xml')//dish" />

  <xsl:template match="/">
    <xsl:apply-templates select="//order" />
    <xsl:text>&#xA;Total: $</xsl:text>
    <xsl:call-template name="total">
      <xsl:with-param name="order" select="//order" />
    </xsl:call-template>
  </xsl:template>

  <xsl:template match="order">
    <xsl:value-of select="@quantity" />
    <xsl:text> x </xsl:text>
    <xsl:value-of select="$dishes[@id = current()/@id]" />
    <xsl:text> ($</xsl:text>
    <xsl:value-of select="$dishes[@id = current()/@id]/@price" />
    <xsl:text>) = $</xsl:text>
    <xsl:value-of select="format-number($dishes[@id =
➥ current()/@id]/@price * @quantity, '0.00')" />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

```

LISTING 17.8 Continued

```
28:   <xsl:template name="total">
29:     <xsl:param name="order" />
30:     <xsl:choose>
31:       <xsl:when test="$order">
32:         <xsl:variable name="subtotal">
33:           <xsl:call-template name="total">
34:             <xsl:with-param name="order"
35                   select="$order[position() != 1]" />
36:           </xsl:call-template>
37:         </xsl:variable>
38:         <xsl:variable name="price"
39:              select="$dishes[@id = $order[1]/@id]/@price" />
40:         <xsl:value-of
41               select="$subtotal + $price * $order[1]/@quantity" />
42:       </xsl:when>
43:       <xsl:otherwise>0</xsl:otherwise>
44:     </xsl:choose>
45:   </xsl:template>
46: </xsl:stylesheet>
```

ANALYSIS Listing 17.8 is used to process Listing 17.7. The data about the dishes in Listing 17.6 is loaded into a variable named `dishes` on line 7. Before going recursive, the stylesheet uses matching on line 10 to display each dish ordered with its price, quantity, and the price multiplied by the quantity. For each `order` element, this is done with the template on line 17. Apart from the template getting the data from each ordered item from the `dishes` variable, not much is going on there. The interesting part of this template is, of course, calculating the total, which is done with the template on line 28. It is called from line 12, with line 13 passing the entire node-set with `order` elements to the `order` parameter. Each time the template calls itself, it takes off one element. This element is added to the total in the same call of the template. The template returns without calling itself again if the node-set is empty. Line 43 makes sure that the value returned in that case is 0. Line 31 checks whether the node-set is empty, and if it's not, line 32 creates a variable named `subtotal`. This variable gets its value from calling the template again. Line 34 passes the parameter, with the selection on line 35 selecting all nodes currently in the node-set, except the node in the first position. It is the node that this template does the calculation for. The subtotal to be returned is created from the subtotal returned from the template call, plus the price of the dish being handled by this template multiplied by the quantity. All these calculations are performed on lines 40 and 41, with the lines before that getting the price from the `dishes` variable. The result is shown in Listing 17.9.

OUTPUT **LISTING 17.9** Result from Applying Listing 17.8 to Listing 17.7

```
1 x Jumbo Prawns ($9.95) = $9.95
2 x Smoked Salmon and Avocado Quesadilla ($10.95) = $21.90
2 x Caesar Salad ($6.95) = $13.90
3 x Grilled Salmon ($19.95) = $59.85
2 x Linguini al Pesto ($16.95) = $33.90
1 x Dame Blanche ($6.95) = $6.95
4 x Banana Split ($6.95) = $27.80

Total: $174.25
```

ANALYSIS Listing 17.9 shows a nice little check, indicating the number of times each dish was ordered, the price of the dish between parentheses, and the subtotal for that dish behind it. Finally, the last line shows the result from totaling with the recursive function.

Summary

In today's lesson, you learned that recursion is a technique to be used when all else fails. This situation usually happens when values have a dynamic feature that can't be dealt with by simple matching and iteration. Using recursion is also a common solution to the drawback that variables can't change while they are in scope.

You also learned that although recursion is elegant, using it everywhere is not a good idea. If you can use matching and iteration, doing so is likely to be much better for performance. This fact is not surprising because this is what XSLT is good at. Recursion can be inefficient because it is implemented using a template that calls itself. Each call costs memory and, unless the compiler is smart, processing cycles as well.

Tomorrow's lesson will pick up where this lesson ends: the computational power (or lack thereof) of XSLT.

Q&A

Q Is there a limit to the number of times a template can call itself?

A Theoretically, no. There is, however, a limit to the memory a processor has available. Because each call costs memory, memory will eventually run out, and the processor will terminate.

Q Can I use all the elements in a recursive template that I can in any other template, matched or called?

A Yes. A recursive template is not different from any other. The only difference is that it calls itself again instead of invoking other templates through matching or calling.

Q Where do I start when I need to make a recursive template?

A Take the problem you have in its smallest form. This is typically just one operation. It should at least be an operation that can be solved without recursion. Create the half of the template that performs this operation. Then change it, so that it calls itself and performs the operation on the half you have and the result of the call. Now create the half that gets processed when there is nothing left to do. That's it. You're done.

Q I've seen that all the recursive templates work with one `xsl:when` and one `xsl:otherwise` element. Can I have multiple `xsl:when` elements?

A Sure. You need at least two alternatives, one recursive and one nonrecursive (the escape hatch). You can, of course, also have more of each.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: You can create recursive templates without parameters.
2. True or False: If there is no "escape hatch," a recursive function will keep calling itself.
3. Can you do recursion with `xsl:apply-templates` instead of `xsl:call-template`?
4. Can a recursive template return a node-set or a tree fragment instead of a simple value?
5. When shouldn't you use recursion?

Exercises

1. Alter Listing 17.4 so that it generates a result such as `Tuco B. P. J. M. Ramirez` instead of `T. B. P. J. M. Ramirez`.

2. Based on Listing 17.3, create a stylesheet that converts

```
<name>John Fitzgerald Kennedy</name>
```

to

```
<name>
  <firstname>John</firstname>
  <firstname>Fitzgerald</firstname>
  <lastname>Kennedy</lastname>
</name>
```

This page intentionally left blank

DAY 18

Building Computational Stylesheets

Yesterday you learned about recursion, a technique you can use when matching and iteration can't do the task you want to perform. Recursion is really a prelude to the topic of today's lesson: stylesheets that perform computations of some sort.

XSLT provides several mathematical operators and functions with which you can accomplish computations. In today's lesson, you will learn about these operators and functions, and how to apply them in real-world problems. Unfortunately, the operators and functions in XSLT are limited in power, so you have to use the existing ones creatively.

In today's lesson, you will learn the following:

- What a computational stylesheet is
- What the difference is between computational stylesheets and other stylesheets
- How to add values of multiple nodes
- How to use operators to perform mathematical functions

Computational Stylesheets Explained

Computational stylesheets are different from most other stylesheets. They serve a different purpose and perform other kinds of transformations.

What Is a Computational Stylesheet?

Most stylesheets transform content from an XML source to some format used for display. In such stylesheets, very often a one-to-one relationship exists between the source document and the resulting document. The data is in the same sequence and basically shows the same structure. When these stylesheets show a different structure, that structure is still very much related to the original structure. This relationship is logical because the main processes are matching and iteration, which closely follow the structure of a source document. These types of stylesheets more or less have a fill-in-the-blanks structure. Formatting is placed in a template, and the template inserts the data from the source document in the places required by the formatting. These stylesheets don't perform any additional operations on the data values of elements and attributes unless they have something to do with formatting, such as capitalizing all characters in a string. This is where computational stylesheets differ most.

Computational stylesheets take the data in the source document, shuffle it around, and use multiple data values to create other values. An example is a stylesheet that transforms sales data into a sales report with aggregate data and so on. The data in the original document is processed to become something totally different. Unlike stylesheets in which the data is mostly reformatted, in a computational stylesheet, there is no way to get the original data back because it's not there anymore. In a stylesheet that just reformatted the data, the data is still there, although it is in another format. For example, say you have two values, 2 and 3. If you format those values to `<b>2</b><b>3</b>`, the separate values can still be retrieved. The only difference with the source document is that the meaning is stripped and is replaced by formatting. Now imagine the values added together. The result of that operation is 5. After the operation is performed, the individual values no longer exist, so there is no way to go back to 2 and 3. That is logical because even if you knew that the result came from two added values, the original values could just as well have been 4 and 1 or 5 and 0.

So, to summarize, computational stylesheets not only transform the data, but also transform the data values. The implication of this definition is that a computational stylesheet doesn't only compute, but might format like a "regular" stylesheet as well. Listing 17.8 in yesterday's lesson, which created a check based on a menu and order data, is a good example of a stylesheet that performs both computation and formatting.

When Do I Use a Computational Stylesheet?

The most common use for a computational stylesheet is to operate on numerical data. Examples of such stylesheets are the previously mentioned sales report, the check in Listing 17.8 from yesterday's lesson, but also a stylesheet creating a ranking based on match results, or a stylesheet solving the traveling salesman problem.

Computational stylesheets don't necessarily operate on number data alone. It is true that this usage is the most common because most real computations involve numerical data. In some situations, the data is not numerical, but the resulting stylesheet is still, in essence, a computational stylesheet. An example of using such a stylesheet is the exercise from yesterday's lesson that transformed

```
<name>Tuco Benedito Pacifico Juan Maria Ramirez</name>
```

into

```
<name>
  <firstname>Tuco</firstname>
  <firstname>Benedito</firstname>
  <firstname>Pacifico</firstname>
  <firstname>Juan</firstname>
  <firstname>Maria</firstname>
  <lastname>Ramirez</lastname>
</name>
```

As you can see, the question "When should I use a computational stylesheet?" doesn't really have a definitive answer. A stylesheet either is or isn't, depending on the type of problem and how you solve it. It is therefore much more interesting to see computational stylesheets in action so that you can see the different tasks you can perform. This lesson therefore contains some elaborate samples.

Operators and Functions Used in Computations

Because all computations depend on operators and functions, you need to know the operations and functions available to you. I discussed some of them in earlier lessons but will discuss them briefly here for completeness.

Note

This section discusses only operators and functions operating on numerical data. The functions that operate on string data were thoroughly discussed on Day 11, "Working with Strings," and will not be discussed again here.

Operators

You've known operators ever since you learned to add 1 and 1 together. When you write down that operation, you get `1 + 1`. The + character in this example is an operator. Operators in XSLT always act on two values. The result depends on what the data type of those two values is. If needed, the values are converted before the operation is performed, so the operation never fails. It might not yield the expected result, but if it is syntactically correct, it will never generate an error.

Caution

Operators (and functions) in XSLT never fail if the syntax is correct, although the result might not be what you want if you made another type of mistake. This is a key feature in XSLT to make sure that stylesheets always produce a result.

The most commonly known operators are mathematical operators, such as plus and minus. Table 18.1 shows the mathematical operators available in XSLT.

TABLE 18.1 Mathematical Operators

Operator	*Usage*	*Result*
`+`	`A + B`	Adds B to A
`-`	`A - B`	Subtracts B from A
`*`	`A * B`	Multiplies A with B
`div`	`A div B`	Divides A by B
`mod`	`A mod B`	Calculates the remainder of dividing A by B a whole number of times

Addition, subtraction, and multiplication are all straightforward. The `div` and `mod` operators, however, need some additional explanation. When you learned division with whole numbers, you learned that A could be divided by B an X number of times. If B doesn't fit exactly an X number of times in A—for instance, 5 divided by 2—you are left with the remainder, or *modulus*. You can calculate the remainder by using the `mod` operator. There is a small difference, though; the `mod` operator in XSLT doesn't require A and B to be whole numbers. A and B can both contain fractions. X, however, is always a whole number, so the remainder is still there if the fit is not perfect.

It seems logical then that the `div` operator divides only by whole numbers, so it is complementary to the `mod` operator. That, however, is not the case. A and B can both contain fractions, as can the result. The reason for this discrepancy lies in the fact that the `div` operator is really used for mathematical operations requiring a result to be sent to the output. The `mod` operator, on the other hand, is mainly used for calculations that have no effect on the values, but on formatting. For instance, `mod` is used on alternating background colors in an HTML table, as shown in Listing 18.1.

LISTING 18.1 Code Using the `mod` Operator to Alternate Colors

```
1:  <xsl:attribute name="bgcolor">
2:    <xsl:choose>
3:      <xsl:when test="position() mod 3 = 0">
4:        <xsl:value-of select="#cc0000" />
5:      </xsl:when>
6:      <xsl:when test="position() mod 3 = 1">
7:        <xsl:value-of select="#00cc00" />
8:      </xsl:when>
9:      <xsl:otherwise>
10:       <xsl:value-of select="#0000cc" />
11:     </xsl:otherwise>
12:   </xsl:choose>
13: </xsl:attribute>
```

ANALYSIS Line 3 in Listing 18.1 checks whether the current node's position within the node-set can be divided by 3. If it can, the `bgcolor` attribute gets the value `#cc0000` on line 4. If it can't, line 6 checks whether the remainder of the division is 1. If that is the case, the `bgcolor` attribute gets the value `#00cc00` on line 7. The last option is that the remainder is 2, in which case the `bgcolor` attribute gets the value `#0000cc` on line 10. It is important here that the `position()` function always returns a whole number. If it doesn't, the checks done in this sample with the `mod` operator don't make much sense.

The preceding sample is *not* computational in nature. In some sense, a computation occurs, but it has bearing only on the formatting of the source document. No changing of values is going on.

Mathematical operators are, of course, intended to be used with number data. This means that if the values used with the operator are not number values, they are implicitly converted. As long as this conversion yields a finite value, you have no problems with any computation. The most common problem here is that the values are converted to `NaN`. This happens when a value is converted from a string value containing characters that aren't allowed in a number, such `abc` or `@`. Providing the source document is correct and you know what you are doing, this is not likely to happen. What is much more likely to happen is that you select a value that is empty or doesn't exist. Selecting such a value is annoying because you would expect that, for instance, `2 +` *`[not existing value]`* would be the same as `2 + 0`. Unfortunately, this isn't the case. The former yields `NaN` as the result because the expression actually says `2 + NaN`. This situation can be extremely annoying because you have to make sure that this never happens, which can be quite a challenge, as you'll see in some samples later in this lesson.

Functions

Operators aren't the only constructs you can use to do computations; you also can use functions. Because XSLT is mostly designed for transformation, the number of functions that are useful in computations is restricted to those for string manipulation discussed thoroughly on Day 10, "Understanding Data Types," and some minor mathematical functions. These functions are dwarfed by the functions available in languages such as C++, Java, and Visual Basic. Basic functions such as determining the square root of a number or sine wave functions are not available in XSLT. This is a serious lack of functionality because it means that you can't do serious calculations, such as needed for graphical data processing or statistical data analysis. With the current state of affairs, you need to look elsewhere or transform to an intermediate format that can be processed to yield the result you want, such as Scalable Vector Graphics (SVG).

Although most mathematical functions in XSLT are minor, you need to know what they are and what they do. Table 18.2 shows a list of the available functions.

Note

Technically, the mathematical functions in XSLT are part of the XPath specification. They are part of XSLT because XPath is used in XSLT to address nodes. Other languages that use XPath, such as XPointer, therefore contain the same functions.

TABLE 18.2 Mathematical Functions

Function	*Description*	*Examples*

`ceiling(`*`value`*`)`	Rounds upward to the closest integer	`ceiling(2.1) = 3`
		`ceiling(2.8) = 3`
		`ceiling(2.0) = 2`
		`ceiling(-2.1) = 2`

`floor(`*`value`*`)`	Rounds downward to the closest integer	`floor(2.1) = 2`
		`floor(2.8) = 2`
		`floor(2.0) = 2`
		`floor(-2.1) = -3`

Function	*Description*	*Examples*

`round(`*`value`*`)`	Rounds to the nearest integer	`round(2.1) = 2`
		`round(2.5) = 3`
		`round(2.8) = 3`
		`round(2.0) = 2`
		`round(-2.1) = -2`

`count(`*`node-set`*`)`	Counts the number of nodes in a node-set	`count(/) = 1` `count(//*)` = number of elements in the document
`sum(`*`node-set`*`)`	Sums the values of the nodes in the node-set	See following discussion

The `sum()` function in Table 18.2 is most interesting to the current discussion. It enables you to select a node-set and sum the values of each node. You can compare this to a spreadsheet application in which you can select a set of values and calculate their sum. A requirement for the `sum()` function is that each node's value can be converted to a number. If one of the nodes contains characters that can't be converted to a number or one of the nodes is empty, the result is `NaN`. If the node-set is empty, the result is zero. This means that the expression `sum(/numbers/number)` used with Listing 18.2 yields 10, `sum(/cars)` yields 0, and `sum(/numbers/*)` results in `NaN`.

LISTING 18.2 Sample Data for the `sum()` Function

```
<numbers>
  <number>2</number>
  <number>3</number>
  <empty />
  <string>abc</string>
  <number>5</number>
</numbers>
```

The `sum()` function is helpful in situations in which you need to total number values. A downside to the `sum()` function is that you can't use it in conjunction with operators. An example will help to clarify this point. Listings 18.3 and 18.4 show the sample data that was used in yesterday's lesson to create a check.

LISTING 18.3 XML Source Representing an "Inventory"

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
```

LISTING 18.3 Continued

```
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note

You can download the sample listings in this lesson from the publisher's Web site.

LISTING 18.4 XML Source with Order Data Related to Listing 18.3

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order id="2" quantity="1" />
  <order id="3" quantity="2" />
  <order id="4" quantity="2" />
  <order id="5" quantity="3" />
  <order id="7" quantity="2" />
  <order id="10" quantity="1" />
  <order id="12" quantity="4" />
</orders>
```

Yesterday you used the sample data in Listings 18.3 and 18.4 to create a check with the dishes ordered and the total. In yesterday's lesson, you used recursion to do the totaling. Listing 18.5 takes an entirely different approach, but the result is the same.

LISTING 18.5 Stylesheet Creating a Check from Listings 18.3 and 18.4

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:    <xsl:variable name="dishes" select="document('18list03.xml')//dish" />
8:
9:    <xsl:template match="/">
10:     <xsl:apply-templates select="//order" />
11:     <xsl:variable name="total">
12:       <xsl:apply-templates select="//order" mode="total" />
13:     </xsl:variable>
14:     <xsl:text>&#xA;Total: $</xsl:text>
15:     <xsl:value-of select="sum($total/subtotal)" />
16:   </xsl:template>
17:
18:   <xsl:template match="order">
19:     <xsl:value-of select="@quantity" />
20:     <xsl:text> x </xsl:text>
21:     <xsl:value-of select="$dishes[@id = current()/@id]" />
22:     <xsl:text> ($</xsl:text>
23:     <xsl:value-of select="$dishes[@id = current()/@id]/@price" />
24:     <xsl:text>) = $</xsl:text>
25:     <xsl:value-of select="format-number($dishes[@id =
➥ current()/@id]/@price * @quantity, '0.00')" />
26:     <xsl:text>&#xA;</xsl:text>
27:   </xsl:template>
28:
29:   <xsl:template match="order" mode="total">
30:     <subtotal>
31:       <xsl:value-of
32:            select="$dishes[@id = current()/@id]/@price * @quantity" />
33:     </subtotal>
34:   </xsl:template>
35: </xsl:stylesheet>
```

ANALYSIS Listing 18.5 contains two templates that match the `order` elements. The first is on line 18, and its job is to output each ordered item with its quantity, price, and the subtotal from multiplying the quantity and price. This is the same as in Listing 17.8 in yesterday's lesson. The template on line 29 matches the `order` element, but in a different mode named `total`. For each `order` element in Listing 18.4, it creates a `subtotal` element. The value of that element is the subtotal for that `order` element, which is calculated on line 32, again by multiplying the quantity with the price of the dish. Line 12 is responsible for invoking the template in `total` mode, the result of which is captured in the `total` variable created on line 11. The result is a node-set that looks like Listing 18.6. Line 15 totals the values of the `total` variable with the `sum()` function.

Note

Under the XSLT 1.0 specification, the code in Listing 18.5 is in error. The `total` variable is a tree fragment, and in XSLT 1.0, a tree fragment can't be converted to a node-set. Because the `sum()` function expects a node-set, this function causes an error in processors that strictly follow the specification, such as MSXML and Xalan. In XSLT 1.1, the conversion is no longer illegal, and some processors, such as Saxon, have changed their implementation to follow XSLT 1.1. Because XSLT 1.1 is not likely to ever reach W3C Recommendation status, existing processors implementing XSLT 1.0 will probably not change their implementation to XSLT 1.1.

LISTING 18.6 Contents of the `total` Variable in Listing 18.5

```
<subtotal>9.95</subtotal>
<subtotal>21.90</subtotal>
<subtotal>13.90</subtotal>
<subtotal>59.85</subtotal>
<subtotal>33.90</subtotal>
<subtotal>6.95</subtotal>
<subtotal>27.80</subtotal>
```

Listing 18.5 is only three-quarters the length of Listing 17.8 in yesterday's lesson. It is much simpler in concept and therefore much simpler to understand. In retrospect, using recursion was like killing a fly with a cannon. I created the contrast on purpose to show you that you can solve the same problem in several ways. In yesterday's lesson, the lack of sequential processing in XSLT was solved with recursion, which, as explained, is the most obvious way to deal with the problem. The solution here is much less obvious, but much simpler and therefore the more desirable method. For small problems like this one, there is not such a performance difference, but with larger datasets, it is likely that recursion is slower. You need to make sure, however, that the solution you create works on the processors you want to use.

Listing 18.5 still seems to have to go in a roundabout way to get the total because the `sum()` function can't aggregate the result from multiplying the price with the quantity. You might want to write something like

```
sum($dishes/@price * /orders/order/@quantity)
```

but this approach doesn't work because the `sum()` function operates on only one node-set and it can't do a nested calculation. So, before aggregating, you need to create a node-set that contains the result of any other operations, which is what the template on line 29 of Listing 18.5 is for, along with capturing the result of the `total` variable on line 11.

The solution in Listing 18.5 and the recursive solution in yesterday's lesson have one thing in common: They both need to calculate the total of the entire check separately from displaying each dish ordered. This means that the stylesheet has to go through the entire order twice: once to display each dish ordered and once to calculate the total. This is rooted to the fact that XSLT has only one output stream, so results are either sent to the output or captured in a variable. This is also linked to the idea that processing should never have side effects that might alter the way data needs to be processed, as explained yesterday. The conclusion is that in XSLT you should perform only one task at a time; not doing so will make your stylesheet much more complex than necessary, if it'll work at all.

Computational Applications

The best way to give you an idea of what kinds of applications you can create with XSLT and which method of attack you can use is by example. This section contains several examples that you might encounter in the real world.

Ranking Teams in a Competition

Suppose you have data from a competition of some kind, for instance, a soccer competition. You can store such data in many different formats, such as tables in a database or a spreadsheet. Another option is storing the data in one or more XML documents and processing it with a stylesheet. The benefit is, of course, that you can store information on teams, matches to be played, and played matches all in XML and use several stylesheets to create output in different formats. Listing 18.7 shows a simple format with teams and played matches.

LISTING 18.7 Sample XML for Ranking Calculation

```
<?xml version="1.0" encoding="UTF-8"?>
<competition>
  <teams>
    <team id="1">Nowhere Losers</team>
    <team id="2">Everywhere Winners</team>
    <team id="3">Somewhere Scorers</team>
    <team id="4">Anywhere Flyers</team>
  </teams>
  <matches>
    <match hometeam="1" awayteam="2" homescore="2" awayscore="2" />
    <match hometeam="3" awayteam="4" homescore="2" awayscore="1" />
    <match hometeam="1" awayteam="3" homescore="3" awayscore="0" />
    <match hometeam="2" awayteam="4" homescore="1" awayscore="4" />
    <match hometeam="4" awayteam="1" homescore="0" awayscore="0" />
    <match hometeam="2" awayteam="3" homescore="3" awayscore="1" />
  </matches>
</competition>
```

ANALYSIS Listing 18.7 consists of two related sets of data. Team information appears on lines 3–8, with each `team` element defining a team with a unique `id` attribute and a team name. The other dataset on lines 9–16 contains the matches that have been played. The `hometeam` and `awayteam` attributes define the teams involved. The values of these attributes are the same as those of the `id` attributes of the `team` elements, so they refer to the `team` elements. The `homescore` and `awayscore` attributes give the points scored by, respectively, the home team and away team. A stylesheet uses this data to determine which team won, and so on. Listing 18.8 shows the stylesheet.

LISTING 18.8 Stylesheet Calculating Team Ranking from Listing 18.7

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:variable name="points">
      <xsl:call-template name="calcpoints">
        <xsl:with-param name="matches"
                        select="/competition/matches/match" />
      </xsl:call-template>
    </xsl:variable>
    <xsl:apply-templates select="$points/points">
      <xsl:sort select="." order="descending" />
      <xsl:sort select="@for - @against" order="descending" />
      <xsl:sort select="@for" order="descending" />
      <xsl:with-param name="teams" select="//team" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="points">
    <xsl:param name="teams" />
    <xsl:value-of select="$teams[@id = current()/@team]" />
    <xsl:text> </xsl:text><xsl:value-of select="." />
    <xsl:text> </xsl:text><xsl:value-of select="@for" />
    <xsl:text> </xsl:text><xsl:value-of select="@against" />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template name="calcpoints">
    <xsl:param name="matches" />
    <xsl:variable name="result">
      <xsl:if test="count($matches) &gt; 1">
        <xsl:call-template name="calcpoints">
          <xsl:with-param name="matches"
                          select="$matches[position() != 1]" />
        </xsl:call-template>
      </xsl:if>
    </xsl:variable>
    <xsl:for-each select="/competition/teams/team">
      <xsl:variable name="id" select="@id" />
      <xsl:variable name="points">
        <xsl:choose>
          <xsl:when test="$result/points[@team = $id]">
            <for><xsl:value-of
                      select="$result/points[@team = $id]/@for" /></for>
            <against>
              <xsl:value-of
                   select="$result/points[@team = $id]/@against" />
            </against>
            <xsl:value-of select="$result/points[@team = $id]" />
          </xsl:when>
          <xsl:otherwise>
            <for>0</for><against>0</against>0
          </xsl:otherwise>
        </xsl:choose>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="$matches[1]/@hometeam = $id">
          <points team="{$id}"
                  for="{$matches[1]/@homescore + $points/for}"
                  against="{$matches[1]/@awayscore + $points/against}">
            <xsl:choose>
              <xsl:when test="$matches[1]/@homescore &gt;
                              $matches[1]/@awayscore">
                <xsl:value-of select="2 + $points/text()" />
              </xsl:when>
              <xsl:when test="$matches[1]/@homescore &lt;
                              $matches[1]/@awayscore">
                <xsl:value-of select="$points/text()" />
              </xsl:when>
              <xsl:otherwise>
                <xsl:value-of select="1 + $points/text()" />
              </xsl:otherwise>
            </xsl:choose>
          </points>
        </xsl:when>
        <xsl:when test="$matches[1]/@awayteam = $id">
          <points team="{$id}"
                  for="{$matches[1]/@awayscore + $points/for}"
                  against="{$matches[1]/@homescore + $points/against}">
            <xsl:choose>
              <xsl:when test="$matches[1]/@homescore &gt;
                              $matches[1]/@awayscore">
                <xsl:value-of select="$points/text()" />
              </xsl:when>
              <xsl:when test="$matches[1]/@homescore &lt;
```

```
89:                                 $matches[1]/@awayscore">
90:                   <xsl:value-of select="2 + $points/text()" />
91:                 </xsl:when>
92:                 <xsl:otherwise>
93:                   <xsl:value-of select="1 + $points/text()" />
94:                 </xsl:otherwise>
95:               </xsl:choose>
96:             </points>
97:           </xsl:when>
98:           <xsl:otherwise>
99:             <xsl:copy-of select="$result/points[@team = $id]" />
100:          </xsl:otherwise>
101:        </xsl:choose>
102:      </xsl:for-each>
103:    </xsl:template>
104: </xsl:stylesheet>
```

Note

Listing 18.8 again uses implicit conversion from a tree fragment to a node-set. Processors that strictly follow XSLT 1.0 will not run this stylesheet. Many processors do offer extension functions for this conversion. Extension functions are the topic of tomorrow's lesson.

ANALYSIS Listing 18.8 is quite lengthy. To keep the discussion focused, I'll discuss it in sections, each time showing only part of the code. But before proceeding with that discussion, look at the result in Listing 18.9.

OUTPUT **LISTING 18.9** Result from Applying Listing 18.8 to Listing 18.7

```
Nowhere Losers 4 5 2
Anywhere Flyers 3 5 3
Everywhere Winners 3 6 7
Somewhere Scorers 2 3 7
```

ANALYSIS In Listing 18.9's output, the teams are ordered by the number of match points they have, which is the first column of numbers. The second column is the number of points scored; and the third column, the number of points scored against the team. If the match points are the same, the second and third columns are also used to order the teams.

Listing 18.8 consists of three templates. The first two templates on lines 7 and 22 are match templates, mainly concerned with display. Note that the template on line 22 uses a parameter to pass team information to it. It is needed because the context matched is a variable, not the source document. This means that the source document is not accessible from the template directly. A global variable or a parameter is needed to access this data. The third template on line 31 is a recursive template that does all the calculations. When it is called, a parameter is passed with all the matches from which it needs to calculate the ranking. The recursive part (lines 33–40 in Listing 18.8) is shown in Listing 18.10.

LISTING 18.10 Recursive Section of the `calcpoints` Template in Listing 18.8

```
1: <xsl:variable name="result">
2:   <xsl:if test="count($matches) &gt; 0">
3:     <xsl:call-template name="calcpoints">
4:       <xsl:with-param name="matches"
5:            select="$matches[position() != 1]" />
6:     </xsl:call-template>
7:   </xsl:if>
8: </xsl:variable>
```

ANALYSIS The code in Listing 18.10 is processed each time the `calcpoints` template is called. Instead of making the recursive call inside an `xsl:choose` element, the call is done inside an `xsl:if` element on line 2, which tests whether the node-set still contains nodes with matches to be processed. This is the case if more than one node is left because the first node will be processed in this recursive call. The result from recursive calls is captured in the `result` variable on line 1. When a recursive call is made, the node-set with matches is sent along, except for the first node. An important detail of this approach is that the `result` variable is empty if no recursive call is made, so when there is only one node left in the node-set in the `matches` variable.

The code between line 40 and line 102 in Listing 18.8 is iterated for `team` in Listing 18.7 so that each time the `calcpoints` template is called, you get an intermediate result, as shown in Listing 18.11.

OUTPUT **LISTING 18.11** Result from Calling the `calcpoints` template

```
<points team="1" for="5" against="2">4</points>
<points team="2" for="6" against="7">3</points>
<points team="3" for="3" against="7">2</points>
<points team="4" for="5" against="3">3</points>
```

ANALYSIS Listing 18.11 is similar to Listing 18.9. All the numbers are the same because this is the actual result from the `calcpoints` template before it is used to display Listing 18.9. For each recursive call, the result is similar to Listing 18.11. When only one match is processed yet, the points for two of the teams are zero.

Line 42 in Listing 18.8 just creates a variable holding the value of the id attribute of the current team being processed. This variable is mainly for quick access to this value, so it is easy to use in comparisons and predicate filtering. Lines 43–58 in Listing 18.8 are responsible for creating a variable named points, which is used to get the current points for the team being processed from the result of a recursive call. This section is shown again in Listing 18.12.

LISTING 18.12 Section of Listing 18.8 Creating a Variable with the Points for the Team Being Processed

```
1:  <xsl:variable name="points">
2:    <xsl:choose>
3:      <xsl:when test="$result/points[@team = $id]">
4:        <for><xsl:value-of
5:                  select="$result/points[@team = $id]/@for" /></for>
6:        <against>
7:             <xsl:value-of
8:                  select="$result/points[@team = $id]/@against" />
9:        </against>
10:       <xsl:value-of select="$result/points[@team = $id]" />
11:     </xsl:when>
12:     <xsl:otherwise>
13:       <for>0</for><against>0</against>0
14:     </xsl:otherwise>
15:   </xsl:choose>
16: </xsl:variable>
```

ANALYSIS Listing 18.12 creates a temporary variable that holds the points for the team currently being processed. If no match has been processed for this team, there is no points element for it yet in the result variable, such as shown in Listing 18.11. As I explained earlier, performing computations with nonexisting values always yields NaN, so you can't do computations directly on the points elements in the result variable because doing so could cause problems. Line 3 tests whether a points element exists for the team being processed in the result variable. If no such element exists, line 13 creates the elements for and against and sets them to zero. This line also inserts the value 0 into the variable directly. This value represents the match points of the team. If a points element exists for the team being processed, the points variable is filled with the values from that element instead. The for and against elements are created on lines 4 and 6, and the match points are inserted on line 10. The computational part of the template can now use the values in the points variable, without having to check whether the team being processed already exists in the result variable because that variable is not used for computational purposes.

If the current team is neither the home team nor the away team for the match being processed in the current recursive template call, line 99 in Listing 18.8 just copies the points element from the `result` variable so that it is available in the result of this template. If the current team is the home team, lines 60–78 in Listing 18.8 compute a new `points` element for this team. This section of the listing is shown again in Listing 18.13.

LISTING 18.13 Section of Listing 18.8 Computing Points for the Current Team from the Current Match

```
1:  <xsl:when test="$matches[1]/@hometeam = $id">
2:    <points team="{$id}"
3:            for="{$matches[1]/@homescore + $points/for}"
4:            against="{$matches[1]/@awayscore + $points/against}">
5:      <xsl:choose>
6:        <xsl:when test="$matches[1]/@homescore &gt;
7:                        $matches[1]/@awayscore">
8:          <xsl:value-of select="2 + $points/text()" />
9:        </xsl:when>
10:       <xsl:when test="$matches[1]/@homescore &lt;
11:                       $matches[1]/@awayscore">
12:         <xsl:value-of select="$points/text()" />
13:       </xsl:when>
14:       <xsl:otherwise>
15:         <xsl:value-of select="1 + $points/text()" />
16:       </xsl:otherwise>
17:     </xsl:choose>
18:   </points>
19: </xsl:when>
```

ANALYSIS Line 1 in Listing 18.13 checks whether the current team being processed matches the home team of the match being processed by the current recursive call. If it matches, line 2 creates a new `points` element and gives the `team` attribute the value of the team's `id` value. Lines 3 and 4 keep track of the points scored and the points scored against the team. Line 6 checks whether the team won the match by checking whether the home team scored more points than the away team. If so, line 8 adds two points to the current points. If the home team lost, line 12 just copies the current value of match points, and if there's a draw, line 15 adds one match point. Lines 79–97 in Listing 18.8 basically do the same thing as Listing 18.13, but then the current team is the away team, not the home team, so the match point calculation is slightly different.

Listing 18.8 is long, specifically the template doing all the calculations. For readability, you might want to break up that template into smaller pieces. Also, note that solutions using the Document Object Model (DOM) with a language such as C++, Java, or Visual Basic are likely to be shorter and simpler. So, why use XSLT to do this calculation in the first place? The answer is basically because you don't need a separate language to perform these calculations. It's not always easy to get the job done, but you can stay within the boundaries of XSLT, which, after all, is meant as a generic language to manipulate XML. When you use the DOM, you have to learn an additional language that can work with the DOM.

String Manipulation

You can use several functions to manipulate strings. Simple tasks, such as filling out a string with spaces so that it is of a certain length, however, can't be performed with those functions alone. You need to create something yourself. You need something to correct text output, such as in Listing 18.9. The columns with numbers aren't aligned because the team names are of different lengths. You could correct this alignment in the source XML, but that's cheating. When you get documents from a third party, you can't use this cheat. Because you can import existing stylesheets, you can create templates that perform this task and import them into other stylesheets. Listing 18.14 shows a stylesheet with two related templates.

LISTING 18.14 Stylesheet with String Functions

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:template name="maxlen">
6:      <xsl:param name="strings" />
7:      <xsl:choose>
8:        <xsl:when test="count($strings) &gt; 1">
9:          <xsl:variable name="result">
10:           <xsl:call-template name="maxlen">
11:             <xsl:with-param name="strings" select="$strings[position()
➥ != 1]" />
12:           </xsl:call-template>
13:         </xsl:variable>
14:         <xsl:choose>
15:           <xsl:when test="$result &gt; string-length($strings[1])">
16:             <xsl:value-of select="$result" />
17:           </xsl:when>
18:           <xsl:otherwise>
19:             <xsl:value-of select="string-length($strings[1])" />
20:           </xsl:otherwise>
21:         </xsl:choose>
22:       </xsl:when>
23:       <xsl:otherwise>
24:         <xsl:value-of select="string-length($strings[1])" />
25:       </xsl:otherwise>
```

```
26:     </xsl:choose>
27:   </xsl:template>
28:
29:   <xsl:template name="insertspaces">
30:     <xsl:param name="number">0</xsl:param>
31:     <xsl:if test="$number &gt; 1">
32:       <xsl:call-template name="insertspaces">
33:         <xsl:with-param name="number" select="$number - 1" />
34:       </xsl:call-template>
35:     </xsl:if>
36:     <xsl:if test="$number &gt; 0">
37:       <xsl:text> </xsl:text>
38:     </xsl:if>
39:   </xsl:template>
40: </xsl:stylesheet>
```

ANALYSIS Listing 18.14 contains two named templates. The `maxlen` template on line 5 returns the length of the longest string in the node-set that is passed to it. The `insertspaces` template on line 29 inserts the given number of spaces. Both are recursive templates, with the latter being quite simple. The `number` parameter on line 30 defines the number of spaces to be inserted. If this number is greater than one, line 32 calls the template again, with the value of the `number` parameter of the current recursive call minus one. If the number is greater than zero, line 37 inserts a space. Line 6 in the `maxlen` template defines the parameter `strings`, which is used to pass the node-set with strings to the template. When the number of strings left is greater than one, line 10 makes a recursive call, sending the node-set on to the next call except for the first node, which is done on line 11. The result is captured in the `result` variable defined on line 9. Line 15 checks whether the length of the current string is larger than the result so far. If the previous result is larger, it is returned with line 16; otherwise, the length of the current string is returned with line 19. If only one string exists, line 24 returns the length of that string. You can see the templates in Listing 18.14 in action when you add Listing 18.15 to it between the `xsl:stylesheet` tags.

LISTING 18.15 Addition to Listing 18.14 Using the `maxlen` and `insertspaces` Templates

```
1:  <xsl:output method="text" encoding="utf-8" />
2:
3:  <xsl:template match="/">
4:    <xsl:variable name="maxlen">
5:      <xsl:call-template name="maxlen">
6:        <xsl:with-param name="strings" select="/competition/teams/team" />
7:      </xsl:call-template>
8:    </xsl:variable>
9:    <xsl:for-each select="/competition/teams/team">
10:     <xsl:value-of select="." />
11:     <xsl:call-template name="insertspaces">
```

LISTING 18.15 Continued

```
12:      <xsl:with-param name="number" select="$maxlen - string-length(.)" />
13:     </xsl:call-template>
14:     <xsl:text> </xsl:text>
15:     <xsl:value-of select="$maxlen - string-length(.)" />
16:     <xsl:text>&#xA;</xsl:text>
17:   </xsl:for-each>
18: </xsl:template>
```

ANALYSIS Listing 18.15 processes Listing 18.7 to show a list with the teams and the number of spaces added. Because of the spaces added, the column of numbers is aligned nicely, as you can see in Listing 18.16. On line 5 of Listing 18.15, the `maxlen` template in Listing 18.14 is called to get the length of the longest team name. The team names are passed to the template on line 6. The result is captured in the `maxlen` variable. On line 9, each `team` element is iterated to create the output. The output consists of the team name, which is output on line 10, followed by the number of spaces that the current team name is shorter than the longest team name. Another space is inserted to separate the output from line 15 from the team name. Line 15 tells how many spaces were inserted by the `insertspaces` template.

OUTPUT **LISTING 18.16** Result from Applying Listing 18.15 to Listing 18.7

```
Nowhere Losers     4
Everywhere Winners 0
Somewhere Scorers  1
Anywhere Flyers    3
```

Converting Currencies

Quite a simple example of a computational stylesheet is a stylesheet performing currency conversion. On the Web, which is inherently international, currency conversion is a fact of life for most e-commerce–type applications. Listing 18.17 shows an XML source that contains the exchange rates for the U.S. dollar to several other currencies.

LISTING 18.17 XML Source with Exchange Rates for the U.S. Dollar

```
<?xml version="1.0" encoding="UTF-8"?>
<currency>
  <rate currency="CAD" name="Canadian Dollar">0.6342</rate>
  <rate currency="EUR" name="Euro">0.8919</rate>
  <rate currency="GBP" name="British Pound">1.4256</rate>
  <rate currency="JPY" name="Japanese Yen">0.00801</rate>
</currency>
```

ANALYSIS Listing 18.17 contains several `rate` elements that define the exchange rate from the U.S. dollar to the currency defined by the three-letter code of the `currency` attribute. Because not everybody is familiar with this code, the `name` attribute contains the full name of the currency.

Listing 18.17 could be provided to e-commerce sites by a bank, for instance, as a Web service. A stylesheet calculating the amount for another currency can use that information. Listing 18.18 is a simple example of such a stylesheet.

LISTING 18.18 Stylesheet Performing Currency Conversions

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:    <xsl:strip-space elements="*" />
7:
8:    <xsl:param name="money">0.0</xsl:param>
9:
10:   <xsl:template match="/">
11:     <xsl:value-of select="concat('US $',$money)" />
12:     <xsl:text> equals&#xA;</xsl:text>
13:     <xsl:apply-templates />
14:   </xsl:template>
15:
16:   <xsl:template match="rate">
17:     <xsl:value-of select="format-number($money div .,'#0.00')" />
18:     <xsl:value-of select="concat(' ',@name,'&#xA;')" />
19:   </xsl:template>
20: </xsl:stylesheet>
```

ANALYSIS Listing 18.18 operates on Listing 18.17 to create a list of amounts from a currency conversion. The amount to be converted is given as a parameter on line 8. The template on line 10 matches the root element of the source document and just outputs some text before using `xsl:apply-templates` on line 13. The template on line 16 matches the `rate` elements in Listing 18.17. Line 17 calculates the amount for the currency being processed, with line 18 telling you which currency it is. The result from applying Listing 18.18 to Listing 18.17 is shown in Listing 18.19.

OUTPUT **LISTING 18.19** Result from Applying Listing 18.18 to Listing 18.17

```
US $49.95 equals
78.76 Canadian Dollar
56.00 Euro
35.04 British Pound
6235.96 Japanese Yen
```

Summary

In today's lesson, you learned that a computational stylesheet is not something you necessarily set out to create. It either is one or isn't one, based on the types of operations you do. You learned that many computations rely on recursion, but that you also can solve many problems with iteration combined with some functions, although you must be careful with some actions that are not supported by all processors. One point to keep in mind is that you always need to perform only one task at a time because your results will likely be wrong if you don't.

XSLT contains several mathematical functions that can help a great deal with computations, most notably the `sum()` function that aggregates the number values of a given node-set. Functions that are somewhat more elaborate don't exist in XSLT, so you need to create them yourself. For square root and sine wave functions, for example, creating them yourself is nearly impossible in XSLT. You can, however, extend the functionality of XSLT with processor extensions, enabling you to create and use functions that are (nearly) impossible in XSLT. Processor extensions are the topic of tomorrow's lesson.

Q&A

Q Other languages have much more computational power. Why would I ever want to use XSLT when it is so ill equipped?

A For major number-crunching applications, you are indeed better off with other languages at present. However, many computations aren't very complex, and having to go through an additional step with another language to perform computations is often impractical. It is much more practical to have one stylesheet doing both computations and transformations.

Q How can I figure out whether I can use the `sum()` function instead of recursion?

A If you can select the numbers you want to aggregate directly with an expression, you're in business. If the data is scattered through a document, it might still be possible, but the expression is probably quite complex. If you need to perform other operations before aggregating, you need to check whether the processor can convert a tree fragment to a node-set.

Q The examples in this lesson all make extensive use of variables and parameters. Is this necessary?

A Yes. Variables are needed to capture intermediate data that can be processed again. Without them, the data would be written to the output unfinished.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: The `sum()` function can't perform operations nested in its argument.
2. True or False: When you use matching on the contents of a variable, the data in the source document is not available.
3. With what expression can you count the number of nodes that are descendents of the current node?
4. Do any operators operate on strings?
5. What is the benefit of performing calculations with XSLT rather than with some other method?

Exercise

1. Create a stylesheet that calculates the ranking of the teams in Listing 18.7 without recursion.

   Hint: Iterate the matches to split them into two elements: one for the home and one for the away team. Capture the output in a variable. Next, iterate the teams and use the `sum()` function to aggregate the elements created for each team. Capture that output in a variable as well, and then iterate and sort it in the same way shown in Listing 18.8.

This page intentionally left blank

DAY 19

Working with XSLT Extensions

In yesterday's lesson, you learned that although you can perform computations with XSLT, creating computations is difficult because of the lack of mathematical functions that you usually find in programming languages. Fortunately, XSLT is extensible, and many processors offer additional functionality that is not part of the XSLT specification itself but can make your life a whole lot easier. Some processors also enable you to write your own extensions.

Today's lesson looks at what extensions are and how you can use them. You also will learn about some common extensions in different processors. Last but not least, you will learn how to create your own extensions.

In today's lesson, you will learn the following:

- What an extension is
- How extensions make stylesheets simpler
- What the drawbacks of extensions are
- How to use common extensions
- How to create your own extensions

Understanding XSLT Extensions

XSLT extensions come into play when XSLT's functionality lacks the capability to do something or when performing a task is so complex in XSLT that the easiest way is to use functionality from outside XSLT. But what exactly are extensions?

What Are XSLT Extensions?

The XSLT specification defines the elements and functions available in XSLT. Basically, if you're talking about XSLT, you're talking about these elements and functions. Although these elements and functions offer you a myriad of possibilities, they are lacking in certain areas, most notably mathematical processing. Fortunately, XSLT is designed to be extensible, so users or vendors can create extensions that provide you with additional functionality that can make complex tasks easy and impossible tasks possible. Extensions exist in three forms:

- Extension elements, which are elements that normally are not part of XSLT
- Extension attributes, which are attributes that normally are not part of the elements defined in XSLT
- Extension functions, which are functions that normally are not part of XSLT

What Are the Benefits of XSLT Extensions?

XSLT extensions provide you with a mechanism to extend the model that XSLT provides. The key benefit of extensions, therefore, is that they enable you to solve problems that are otherwise impossible to solve or take an immense amount of code. Especially in mathematical processing, you can extend XSLT so that you can use the basic mathematical functions available in other languages. In these languages, such functionality often requires no more than just one or two lines of code, whereas the same operation in XSLT can require tens or even hundreds of lines of code, if the functionality can be achieved with pure XSLT in the first place. A simple function such as calculating the square root of a number is nearly impossible to create with pure XSLT. The algorithm for this function is quite complex. Most languages, such as C++, Java, and Visual Basic, however, have this function built in, and invoking it from XSLT takes just a few lines of code. In fact, invoking this function is so simple that you might question the decision to leave it out of XSLT in the first place. However, that's the way things are, so you need to deal with them.

Another major benefit of XSLT extensions, probably the reason they were created to begin with, is that they enable you to create applications that are very specific to a certain area—for instance, applications dealing with chemistry. Just like you can define vocabularies that are specifically suited for these application domains, you can create extensions that perform functions specific to that domain, possibly on the basis of the vocabulary that you created. On Day 7, “Controlling the Output,” you learned about applications that output Portable Document Format (PDF) or Rich Text Format (RTF). These applications can be implemented as extensions to XSLT, so you can create PDF or RTF documents straight from the XSLT processor, instead of having to use an additional application or component in your application.

The Drawbacks of XSLT Extensions

XSLT extensions are handy, but they have one major drawback: Extensions are processor specific. This means that if you create an extension for Saxon, the same extension can’t be used with Xalan or Microsoft’s MSXML component. This incompatibility tosses the concept of cross-platform, cross-processor XSLT right out the window. Any stylesheet that uses specific extensions implemented on a specific parser can’t be run with any other processor. XSLT 1.1 addresses this drawback for extension functions up to a certain degree: With XSLT 1.1, the `xsl:script` element enables you to create extension functions within a stylesheet using ECMAScript, JavaScript, or Java. With the same element, you also can reference a separate source document that defines an extension function.

Note

XSLT 1.1 development has been abandoned, so it will probably never reach W3C Recommendation status. Some processors, such as Saxon, have implemented several XSLT 1.1 features, some of which differ from XSLT 1.0.

Some processors currently offer an extension element that serves the same function as the XSLT 1.1 `xsl:script` element. You can use the vendor-specific `script` elements to create extension functions. Because these elements are processor specific, you do need to create a function multiple times, one for each processor. You also have to check which one you should use at runtime. In the following sections, you will learn how to do both of these things.

Tip

Sticking to XSLT as much as possible is always a good idea. If you can solve a problem without resorting to extensions, go for it! Use extensions only if you can save a lot of time or if you can’t achieve something by using pure XSLT.

Another point to keep in mind is that programming with extensions isn’t always as easy as it looks. Certain coding mistakes can cause your whole stylesheet to produce no output at all. You just draw a blank, no errors, nothing. This type of response isn’t true for every type of mistake and probably not with all processors, but it is something to look out for because tracking the problem in these cases is very hard. If you still decide to use extensions, start with something simple to check whether the result is what you want. Slowly build your stylesheet to make sure that everything works. If you get into a situation in which the stylesheet ceases to function, you know that the last change you made caused the error, so you know where to look.

How Do XSLT Extensions Work?

XSLT extensions operate separately from the core XSLT elements and functions. To achieve this separation, extension elements, attributes, and functions need to use a separate namespace. When the processor encounters an element, attribute, or function that is part of that namespace, the function is invoked more or less as if it is part of XSLT. When this situation occurs with a function defined within the stylesheet itself with a `script` element, the processor simply needs to invoke that script. Extension elements, attributes, or functions that depend on an external code library are somewhat trickier because the processor needs to know how to invoke that code library. With products such as Saxon, the value of the namespace name determines how to invoke the code library. Specifically, elements need to implement a specific interface so that the processor knows how to call it.

Note

Building your own extension elements is a complex task requiring knowledge and experience with programming and the underlying platform. It is beyond the scope of this book to discuss creating extension elements. This book will discuss only creating your own extension functions with the `script` element.

Using Built-in Extensions

To distinguish extensions from regular XSLT constructs, the elements, attributes, or functions need to use a namespace that is separate from the namespace used by XSLT. You need to declare this namespace in the `xsl:stylesheet` element. In addition, you need to add the namespace prefix of the declared namespace to the value of the `extension-element-prefixes` attribute. This attribute tells the processor that the given namespace prefix or prefixes denote extension elements, attributes, or functions. The value of the attribute should be a whitespace-separated list of namespace prefixes. The following `xsl:stylesheet` element adds Saxon extensions to the stylesheet:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://icl.com/saxon"
  extension-element-prefixes="saxon">
```

The prefix `saxon` is used for the Saxon extension. Because this is a regular namespace, you can use any other nondefault prefix, as long as the namespace name is the same. If you want to use extensions with a different processor—for instance, MSXML—the value of the namespace name is different. For MSXML, the stylesheet start tag should look like this:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt">
```

Microsoft has chosen to use a Uniform Resource Name (URN) instead of a Uniform Resource Locator (URL) to declare the namespace. The principle is the same, however. Also, note that the `extension-element-prefixes` attribute is not defined. If you want to do everything by the book, you should add it, but MSXML detects that you're working with extensions itself. Because most processors require the `extension-element-prefixes` attribute, adding it anyway is wise. Otherwise, if you change your stylesheet to another processor later, you'll have a hard time figuring out why it isn't working. In that case, the most likely result is that the extension elements are inserted into the result as is rather than processed.

Finally, if you want to use extensions provided by Xalan, the start tag of the stylesheet should look like this:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xalan="http://xml.apache.org/xalan"
  exclude-result-prefixes="xalan">
```

Xalan also doesn't require the `extension-element-prefixes` attribute, but only for the built-in extensions. Note that the `xalan` namespace prefix is also excluded from the output explicitly with the `exclude-result-prefixes` attribute. If you don't declare such an attribute, Xalan is likely to declare the namespace in any XML output. To avoid such side effects, you would be wise to add the `exclude-result-prefixes` attribute with all the namespaces declared for processor extensions so that they'll never appear in any output.

Using Extension Elements

Extension elements are just like regular XSLT elements, with the exception, of course, that they are not part of the XSLT language and can be used only with specific processors. Other than that, their usage is more or less the same. Saxon provides quite a few extension elements. All of them are described in the product documentation that comes with Full Saxon.

Note

Discussing all the extension elements in Saxon is beyond the scope of this book.

The extension elements in Saxon are discussed in the file named extensions.html, which you can find in the doc directory of the Saxon installation directory. You can also find this documentation at `http://users.iclway.co.uk/mhkay/saxon/saxon6.2.2/extensions.html`. Documentation on the extensions in MSXML is part of the MSXML SDK, which you can download from `http://msdn.microsoft.com/xml/`. Xalan's extensions are described in the file named extensions.html in the `doc` directory of the installation, and can also be found at `http://xml.apache.org/xalan-j/extensions.html`.

One extension element in Saxon that could have benefited examples in preceding lessons is the `saxon:assign` element. This element enables you to change the contents of a variable, getting around the problem that variables in XSLT can't change while they are in scope. To see how this element can benefit you, let's look at an example that creates a check just like examples from the preceding lessons. Listings 19.1 and 19.2 show the source XML documents.

LISTING 19.1 XML Source Representing an "Inventory"

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <appetizers title="Work up an Appetite">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon and Avocado Quesadilla</dish>
    <dish id="4" price="6.95">Caesar Salad</dish>
  </appetizers>
  <entrees title="Chow Time!">
    <dish id="5" price="19.95">Grilled Salmon</dish>
    <dish id="6" price="17.95">Seafood Pasta</dish>
    <dish id="7" price="16.95">Linguini al Pesto</dish>
    <dish id="8" price="18.95">Rack of Lamb</dish>
    <dish id="9" price="16.95">Ribs and Wings</dish>
  </entrees>
  <desserts title="To Top It Off">
    <dish id="10" price="6.95">Dame Blanche</dish>
    <dish id="11" price="5.95">Chocolat Mousse</dish>
    <dish id="12" price="6.95">Banana Split</dish>
  </desserts>
</menu>
```

Note: You can download the sample listings in this lesson from the publisher's Web site.

LISTING 19.2 XML Source with Order Data Related to Listing 19.1

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order id="2" quantity="1" />
  <order id="3" quantity="2" />
  <order id="4" quantity="2" />
  <order id="5" quantity="3" />
  <order id="7" quantity="2" />
  <order id="10" quantity="1" />
  <order id="12" quantity="4" />
</orders>
```

Listing 19.3 shows a stylesheet that uses the `saxon:assign` extension function. Because it uses this function, there is no need for recursion or a separate iteration to total the result.

LISTING 19.3 Stylesheet Using Saxon Extension Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://icl.com/saxon"
  extension-element-prefixes="saxon"
  exclude-result-prefixes="saxon">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />
  <xsl:variable name="dishes" select="document('19list01.xml')//dish" />
  <xsl:variable name="total" select="0" saxon:assignable="yes" />

  <xsl:template match="/">
    <xsl:apply-templates select="//order" />
    <xsl:text>&#xA;Total: $</xsl:text>
    <xsl:value-of select="$total" />
  </xsl:template>

  <xsl:template match="order">
    <xsl:variable name="price"
                  select="$dishes[@id = current()/@id]/@price" />
    <xsl:value-of select="@quantity" />
    <xsl:text> x </xsl:text>
```

LISTING 19.3 Continued

```
24:     <xsl:value-of select="$dishes[@id = current()/@id]" />
25:     <xsl:text> ($</xsl:text>
26:     <xsl:value-of select="$price" />
27:     <xsl:text>) = $</xsl:text>
28:     <xsl:value-of select="format-number($price * @quantity, '0.00')" />
29:     <xsl:text>&#xA;</xsl:text>
30:     <saxon:assign name="total" select="$total + $price * @quantity" />
31:   </xsl:template>
32: </xsl:stylesheet>
```

ANALYSIS Listing 19.3 uses the `saxon:assign` element to alter the value of a global variable that holds the total of the entire order up to the point that the orders are processed. For each order, the subtotal is added to the total. Because the stylesheet uses Saxon extensions, line 4 declares the namespace for those extensions. Line 5 uses the `extension-element-prefixes` attribute to tell the processor that the declared namespace is used for extensions. The `exclude-result-prefixes` attribute makes sure that if the output is XML (which, in this case, it isn't), the namespace used for the extensions is not copied to the result document. The global variable `total` is defined on line 11. To tell Saxon that the variable can be changed during processing, you add the `saxon:assignable` attribute with the value `yes`. If you don't add this attribute, Saxon can't change the value of the variable. The template on line 13, which matches the root element, does nothing else but invoke matching for the `order` elements on line 14, output some text, and output the final value of the `total` variable on line 15. The template on line 19, which matches the `order` elements, displays each order with the name, the number of times it was ordered, its price, and the subtotal for the dish. To make it all a bit easier to work with, line 20 creates a `price` variable with the price of the dish being processed. Line 30 uses the `saxon:assign` element to give the `total` variable a new value. This value is its previous value plus the subtotal for the dish being processed, calculated from the price and quantity. So, for each dish, the subtotal is added to the total. The result of this stylesheet is the same as that in Listing 18.5 and Listing 17.8, which are based on other types of calculations. The result is shown in Listing 19.4.

OUTPUT **LISTING 19.4** Result from Applying Listing 19.3 to Listing 19.2 in Saxon

```
1 x Jumbo Prawns ($9.95) = $9.95
2 x Smoked Salmon and Avocado Quesadilla ($10.95) = $21.90
2 x Caesar Salad ($6.95) = $13.90
3 x Grilled Salmon ($19.95) = $59.85
2 x Linguini al Pesto ($16.95) = $33.90
1 x Dame Blanche ($6.95) = $6.95
4 x Banana Split ($6.95) = $27.80

Total: $174.25
```

ANALYSIS As you can see from the output, using an extension element has no influence on the result. Programming Listing 19.3 is much easier than any of the other solutions, but with the disadvantage that it works only with Saxon.

Using Extension Functions

Extension functions don't differ much from functions supported by XSLT by default. The difference is that these functions are defined for specific processors and might not conform completely to the goals of XSLT. XSLT is designed to be free of side effects, so it could be processed in parallel or incrementally (as discussed on Day 17, "Using Recursion"). With extension functions, you have no guarantee that they have no side effects. Saxon supports more than 25 extension functions, some of which are quite useful. One of the extension functions is `saxon:sum()`, which is similar to the `sum()` function provided by XSLT. The major difference is that the `saxon:sum()` function is not restricted to aggregating the value in the given node-set; it can also be provided with an expression to be used on the nodes in the node-set before the aggregation is done. The result of the expression executed for each node in the node-set is aggregated. Listing 19.5 shows how you can use this function to create yet another stylesheet to calculate the check from Listings 19.1 and 19.2.

LISTING 19.5 Stylesheet Using Saxon Extension Functions

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://icl.com/saxon"
  extension-element-prefixes="saxon"
  exclude-result-prefixes="saxon">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />
  <xsl:variable name="dishes" select="document('19list01.xml')//dish" />

  <xsl:template match="/">
    <xsl:call-template name="displayorder">
      <xsl:with-param name="order">
        <xsl:apply-templates select="//order" />
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>

  <xsl:template match="order">
    <item price="{$dishes[@id = current()/@id]/@price}">
     <xsl:copy-of select="@*" />
      <xsl:value-of select="$dishes[@id = current()/@id]" />
    </item>
```

LISTING 19.5 Continued

```
25:   </xsl:template>
26:
27:   <xsl:template name="displayorder">
28:     <xsl:param name="order" />
29:     <xsl:for-each select="$order/item">
30:       <xsl:value-of select="@quantity" />
31:       <xsl:text> x </xsl:text>
32:       <xsl:value-of select="." />
33:       <xsl:text> ($</xsl:text>
34:       <xsl:value-of select="@price" />
35:       <xsl:text>) = $</xsl:text>
36:       <xsl:value-of select="format-number(@price * @quantity, '0.00')" />
37:       <xsl:text>&#xA;</xsl:text>
38:     </xsl:for-each>
39:     <xsl:text>&#xA;Total: $</xsl:text>
40:     <xsl:value-of select="saxon:sum($order/item,saxon:expression('@price
➥ * @quantity'))" />
41:   </xsl:template>
42: </xsl:stylesheet>
```

ANALYSIS The first 10 lines of Listing 19.5 are the same as those in Listing 19.3. These lines declare the necessary extension namespaces and so on. The root element of the source document is matched by the template on line 12. This template calls the `displayorder` template, which creates the output for all the orders and the total. This output is created from the `order` parameter passed on line 14. Note that instead of creating a variable and passing it to the template in a subsequent template call, an `xsl:apply-templates` element is nested in the `xsl:with-param` element to give the parameter its value. Using this approach, you don't need to create an additional variable because the contents of the parameter are created directly. The resulting parameter looks like Listing 19.6. What happens in the `displayorder` template should by now be familiar to you, except for line 40, which calculates the total using the `saxon:sum()` function. This function has two arguments: one passing the node-set that needs to be aggregated and one with an expression that defines what the processor needs to do with each node before it is added to the aggregate. The second argument uses the `saxon:expression()` function to define the expression to be used. In this case, the expression tells the processor that what it needs to aggregate for each node is the `quantity` attribute multiplied by the `price` attribute.

LISTING 19.6 Value of the Parameter Passed to the `displayorder` Template in Listing 19.5

```
<item price="9.95" id="2" quantity="1">Jumbo Prawns</item>
<item price="10.95" id="3" quantity="2">Smoked Salmon and Avocado
➥ Quesadilla</item>
<item price="6.95" id="4" quantity="2">Caesar Salad</item>
<item price="19.95" id="5" quantity="3">Grilled Salmon</item>
<item price="16.95" id="7" quantity="2">Linguini al Pesto</item>
<item price="6.95" id="10" quantity="1">Dame Blanche</item>
<item price="6.95" id="12" quantity="4">Banana Split</item>
```

Because of the way the `saxon:sum()` function works, the expression passed to it can have bearing only on the node-set itself and nodes that are directly addressable from each node. So, you can't use an expression such as

```
@quantity * $dishes[@id = current()/@id]/@price
```

The `current()` function doesn't work inside the expression because the expression created is more or less static. The `current()` function would therefore be evaluated once, when the expression is created. It is not evaluated for each node of the node-set, so the result is always the same. To get around this problem, Listing 19.5 first creates a parameter where both the `price` and `quantity` attributes are included so that they can be addressed through the static expression Saxon creates.

Using Extensions with Other Processors

In the preceding sections, you learned how to use extension elements and extension functions with the Saxon processor. With other processors, the procedure is similar. The main differences lie in how the namespace needs to be declared. Also, each processor has different extension functions and elements. Those supported by Xalan, for instance, differ a great deal from those offered by Saxon. MSXML offers only one extension element, `msxsl:script`, and one extension function, `msxml:node-set()`, which converts a tree fragment to a node-set with a single node. This function exists in Xalan and Saxon as well, although its name is slightly different in Saxon.

Creating Your Own Extension Functions

In the preceding section, you learned how to use built-in extensions. Even if the processor supports extensions, chances are they don't do what you want. For instance, if you need to calculate the square root of a number, you're out of luck because no extension function performs this task. You can create your own extension functions, however, either by using a script embedded in a stylesheet or by referencing functionality that lies outside the processor.

Using Java Functions as Extension Functions

Java-based processors such as Saxon, Xalan, xt, and Oracle XSL enable you to reference Java classes and use their functionality. This capability is immensely powerful because you can perform functions on the data in the source document and also perform functions with the data from the source document—for instance, sending e-mail or updating a database. You can compare this functionality with browser plug-ins that enable you to view content that you otherwise can't or to use some interactive medium. In essence, this functionality enables you to create data-driven applications centered around XML data and the XSLT programming language.

Note

From MSXML, you can't reference Java functions. The only way to use functionality that lies outside the processor is through code in the `msxsl:script` extension element. This topic will be discussed later in this lesson.

Using Java classes and functions from Saxon or Xalan is remarkably easy. You have to declare a namespace that incorporates the Java class that you want to use. For instance, if you want to use the full range of mathematical functions provided by Java, you can declare the following namespace:

```
xmlns:math="java.lang.Math"
```

This references the `java.lang.Math` class, so you can use the functions available in that class. You should also incorporate the `extension-element-prefixes` and `exclude-result-prefixes` attributes to deal with this namespace properly.

Note

According to the Xalan documentation, the preceding namespace declaration is not the preferred method. Instead, you should use `xalan://java.lang.Math` as the namespace name, after including a special `xalan` namespace declaration. Both Xalan and Saxon use the class name specified after the last slash, however, and ignore everything that precedes it. Using the preceding declaration is therefore the simplest way to ensure that both Saxon and Xalan can run the same stylesheet.

Listing 19.7 shows a simple sample document with some numbers that you can use with some mathematical functions.

LISTING 19.7 Sample XML with Numbers to Be Processed

```
<?xml version="1.0" encoding="UTF-8"?>
<numbers>
  <number>2</number>
  <number>9</number>
  <number>144</number>
  <number>65536</number>
  <number>123456789</number>
</numbers>
```

The simple stylesheet in Listing 19.8 demonstrates the use of a Java class as a processor extension. From Listing 19.7, it creates a list that tells you the square root of each number.

LISTING 19.8 Stylesheet Using Java Function

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:math="java.lang.Math"
5:    extension-element-prefixes="math"
6:    exclude-result-prefixes="math">
7:
8:    <xsl:output method="text" encoding="utf-8" />
9:
10:   <xsl:template match="/">
11:      <xsl:for-each select="numbers/number">
12:       <xsl:text>The square root of </xsl:text>
13:       <xsl:value-of select="." />
14:       <xsl:text> is </xsl:text>
15:       <xsl:value-of select="math:sqrt(.)" />
16:       <xsl:text>.&#xA;</xsl:text>
17:     </xsl:for-each>
18:   </xsl:template>
19: </xsl:stylesheet>
```

ANALYSIS Listing 19.8 calculates the square root for each number in Listing 19.7. To do so, the stylesheet uses the `sqrt()` function that is part of the `java.lang.Math` class. A reference to this library is created on line 4, which declares the `math` namespace. Lines 5 and 6 make sure that this namespace is seen as a processor extension and that this namespace is excluded from any output. The stylesheet contains only one template, which iterates through the numbers in Listing 19.7 on line 11. Line 15 is the most interesting of each iteration; the rest is just formatting. Line 15 uses `math:sqrt()` to calculate the square root of the current number. Listing 19.9 shows the result of Listing 19.8.

OUTPUT **LISTING 19.9** Result from Applying Listing 19.8 to Listing 19.7 in Saxon

```
The square root of 2 is 1.4142135623730951.
The square root of 9 is 3.
The square root of 144 is 12.
The square root of 65536 is 256.
The square root of 123456789 is 11111.111060555555.
```

The preceding example shows that using Java classes as extensions is easy. The example, however, is a little deceiving because it works with a simple data type. The situation becomes a little more complex when the argument sent to a class is a node-set or tree fragment. Unless the Java class can figure out how to process the value, chances are the class will not do the job it should do. In fact, it will probably fail, without telling you what went wrong, because the result is just a big blank.

A nice feature of the Java extension mechanism is that when the value passed to the class is a node-set, it can be accessed as a `nodelist` in the Document Object Model (DOM). In addition, you can write your own Java classes and use them as extensions. Because the data conforms to the XML DOM, accessing the individual values in the node-set is relatively painless.

Note Demonstrating how to create your own Java class and use it as an extension is beyond the scope of this book because this procedure requires experience with both Java and DOM.

Creating an Extension Function with Script

One of the most exciting features in XSLT is the capability to create your own extensions using script embedded in a stylesheet. As I said before, this procedure currently is implemented with processor extension elements, but this will likely change, at which point this feature will be platform and processor independent.

The concept is simple. You create a `script` element that holds ECMAScript, Java, or JavaScript code. This code can be called more or less like an extension function of the processor or a Java class. The downside to both of those mechanisms is that your processor needs to support the chosen language. In the case of Java classes, this means that the Java runtime and classes must be available to the processor. With the script embedded in the stylesheet, Java support is not necessary , but the processor must be able to run the script code. The processor either needs to have an embedded script processor or needs to call an external script processor.

The script functions you create can access the data of the arguments through the DOM, the same as the Java extensions discussed in the preceding section.

The best implementation of this feature is in the MSXML component. With Saxon, this element currently works only with external scripts, and with Xalan, this feature is more complex. In addition, Saxon's documentation doesn't give you much information to go on. Both Saxon and Xalan clearly favor the approach of writing extension functions as external Java classes, as discussed in the preceding section. Because MSXML doesn't support Java, it relies on the `msxsl:script` element to extend the processor's functionality with script. From the script, you can call COM components, which serve the same purpose as Java classes. Windows contains many COM components, but you can also create them yourself by using languages such as Visual C++, Visual Basic, and Delphi. If the Java runtime is installed on Windows, you can also register Java classes as COM components. Be aware that COM components are native to Windows, however, so they can't be used on other platforms. Some tests indicate that MSXML script extensions are up to 30 times faster than Java extensions in Saxon or Xalan. Once a new XSLT standard supports a platform-independent extension method, such as embedded script, this will be the preferred solution because it will work on any platform and processor. Java extensions, on the other hand, require Java-based or Java-enabled processors, and COM-based extensions will work only on Windows. The performance versus portability aspect of XSLT extensions in Java and COM is shown in Figure 19.1

FIGURE 19.1

Portability versus performance with XSLT extensions.

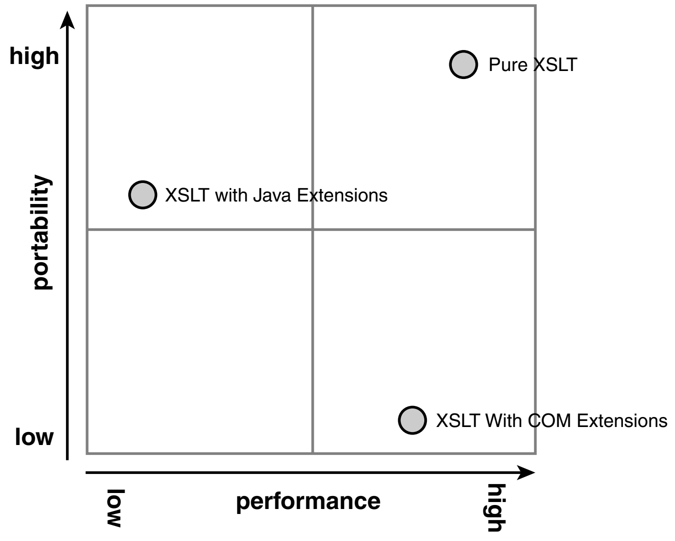


Figure 19.1 shows you that pure XSLT performs better than XSLT with extensions of any kind. This is logical because the XSLT processor is specifically designed to perform XSLT transformations as quickly as possible. Pure XSLT is also the most portable, although still not 100% portable, because of processor differences. COM extensions perform very well but are restricted to the Windows platform and therefore not very portable. Java extensions, on the other hand, work on any platform/processor that supports Java. Java extensions do not perform well, however.

When you use the `msxsl:script` element, you need to tell the processor which language is used to implement the functions with the `language` attribute. For MSXSL, this is any valid Active Scripting Language, which is any scripting language that is available for the Windows Scripting Host. JavaScript and VBScript are available by default, but other languages are available from third parties, such as Perl from `http://www.activestate.com/`. You can find more information about Microsoft scripting language support at `http://msdn.microsoft.com/scripting`. Besides defining the language used, you also need to tell the processor which namespace these functions will be part of. You can use any namespace you like; the namespace name doesn't have to point to any specific location. You attach this namespace to the functions inside the `msxsl:script` element with the `implements-prefix` attribute, which can contain only one namespace prefix. So, a proper `msxsl:script` element looks like this:

```
<msxsl:script language="javascript" implements-prefix="user">
```

This line defines an `msxsl:script` element that uses JavaScript to implement the functions. The functions are available under the `user` namespace prefix.

The sample document in Listing 19.10 is based on the match calculation example in yesterday's lesson.

LISTING 19.10 Sample XML with Different Length Strings

```
<?xml version="1.0" encoding="UTF-8"?>
<teams>
  <team id="1">Nowhere Losers</team>
  <team id="2">Everywhere Winners</team>
  <team id="3">Somewhere Scorers</team>
  <team id="4">Anywhere Flyers</team>
</teams>
```

ANALYSIS Listing 19.10 shows several teams. The team names are of interest for this example. Listing 18.15 in yesterday's lesson contained two templates to fill out the team names, so the number columns would line up properly. Listing 19.11 does the same thing using embedded script functions.

LISTING 19.11 Stylesheet with Embedded Script Extension Functions

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:        xmlns:msxsl="urn:schemas-microsoft-com:xslt"
5:        xmlns:user="http://www.aspnl.com/xmlns/extensions">
6:
7:    <msxsl:script language="javascript" implements-prefix="user">
8:       function maxlen(nodelist) {
9:         var len = 0;
10:        for (var i=1; i &lt; nodelist.length; i++) {
11:          var s = new String(nodelist.nextNode().text);
12:          if(len &lt; s.length) {
13:            len = s.length;
14:          }
15:        }
16:        return len;
17:      }
18:
19:      function inschars(num,ch) {
20:        var s = '';
21:        for (var i=0; i &lt; num; i++) {
22:          s = s + ch;
23:        }
24:        return s;
25:      }
26:   </msxsl:script>
27:
28:   <xsl:output method="text" encoding="utf-8" />
29:
30:   <xsl:template match="/">
31:     <xsl:variable name="maxlen" select="user:maxlen(/teams/team)" />
32:     <xsl:for-each select="/teams/team">
33:       <xsl:value-of select="." />
34:       <xsl:value-of select="user:inschars($maxlen -
➥ string-length(.),'-')" />
35:       <xsl:text> </xsl:text>
36:       <xsl:value-of select="$maxlen - string-length(.)" />
37:       <xsl:text>&#xA;</xsl:text>
38:     </xsl:for-each>
39:   </xsl:template>
40: </xsl:stylesheet>
```

ANALYSIS Listing 19.11 uses extension functions, so the namespaces that they are part of need to be declared properly. The `msxsl:script` element is used on line 7 to create the extension functions, so that namespace needs to be declared first, which is done on line 4. Line 5 declares a namespace with the prefix `user`, which is the namespace of the functions created in the script. Line 7 tells you that the functions implemented are for this namespace and that the language used is JavaScript. The function `maxlen` on line 8 returns the length of the largest string in a node-set. The argument needs to be a node-set, or the function will fail. Line 9 sets the return value to `0`, and line 10 creates a loop to iterate through all the nodes in the node-set. Note that the number of iterations is determined from the `length` property of the `nodelist` argument. This argument is a DOM `nodelist` and is fully accessible with DOM functionality. This is visible again on line 11, which moves to the next node in the node-set and places the string value in the `s` variable. Line 12 checks whether the length of this string is longer than the current maximum length, and if so, line 13 changes it. Note that the `length` property here is part of the string object and has nothing to do with DOM functionality. Finally, line 16 returns the length of the longest string.

The function `inschars` on line 19 is a variation of the `insertspaces` template in Listing 18.15. This function, which inserts the character that is passed to it a given number of times, has no DOM functionality because all the arguments are simple types. The function simply defines an empty string, to which each iteration on line 21 adds the given character.

The template on line 30 gets the length of the longest team name on line 31, iterates through the team names on line 32, outputs the team name, and then inserts the hyphen character to fill out the team name. The result is shown in Listing 19.12.

OUTPUT **LISTING 19.12** Result from Applying Listing 19.11 to Listing 19.10 in MSXSL

```
Nowhere Losers---- 4
Everywhere Winners 0
Somewhere Scorers- 1
Anywhere Flyers--- 3
```

ANALYSIS Listing 19.12 shows that the `inschars` function in Listing 19.11 inserts hyphen characters if the team name is shorter than the longest one. The space is inserted by line 35 of Listing 19.11.

Summary

In today's lesson, you learned that XSLT is extensible. This feature enables users and vendors to add functionality to XSLT to make difficult tasks easy and impossible tasks possible. To distinguish the additional functionality from native XSLT functionality, the added elements, attributes, or functions are part of a separate namespace. Most processors offer a number of extension elements and functions that you can use in a stylesheet.

Because the functions are provided by the different processors, a stylesheet using these functions is no longer processor independent and possibly no longer platform independent.

Many Java-based processors enable you to call functions from Java classes through the extension mechanism. This capability is very powerful because you can create your own Java classes and call them from within the stylesheet. These functions can perform simple tasks on the data or, for example, send an e-mail or update a database. MSXML exhibits this same power through script that is embedded in the stylesheet, which can call COM components that are available on the system, or which you have created with a COM-enabled programming language.

A feature that will be available in future versions of XSLT enables you to create your own processor-independent extension functions. Using ECMAScript, Java, or JavaScript embedded in the stylesheet, you can perform operations that are not available in XSLT itself. XSLT 1.0 doesn't offer this capability, but the major processors all have an extension element with similar capabilities.

In tomorrow's lesson, you will learn more details about the differences between the different processors and how you can deal with these differences—for instance, how to deal with different processor extensions.

Q&A

Q If I create an extension function, do I have access to the entire document and all the variables?

A No. An extension function has access only to the data that is passed to it. Any data you want to get needs to be addressable from that data.

Q Can extension elements or functions have side effects?

A Unfortunately, yes. Nothing guarantees that they are free from side effects. In fact, the `saxon:assign` element is an example of an element that, from an XSLT perspective, has side effects because it makes the result nondeterministic.

Q Do all processors support extension elements and functions?

A Not necessarily. Supporting extension elements and functions isn't really a requirement, so some vendors may elect not to add extensions and the capabilities to add them yourself. One extension function that is implemented by most processors is a function that converts a tree fragment to a node-set. As explained in yesterday's lesson, that action is not possible in XSLT 1.0. In XSLT 1.1, it is possible, but most processors will not implement XSLT 1.1 because it is not likely to ever become a W3C Recommendation.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Extension elements are processor independent.
2. True or False: The `extension-element-prefixes` attribute is required when you want to use extension functions.
3. What do you need to do to use functions in a Java class from a Java-based or Java-enabled processor?
4. In what situation can't you use extensions?
5. Why would you want to create a function embedded in a stylesheet when that same function is available as an extension function for the processor?

Exercises

1. Create a stylesheet for Saxon that uses the `saxon:max()` function to determine the largest number in Listing 19.7.

   Hint: The argument of the `saxon:max()` function should be a node-set containing the numbers of which you want the maximum number.
2. Create a stylesheet from MSXML that uses a `max()` function that you define within the stylesheet by using the `msxsl:script` element of the processor.

   Hint: The `script` function has many similarities with the `maxlen` function in Listing 19.11. If you can't figure out what the script is supposed to look like, take a sneak peek at the solution and try to solve the rest yourself.

DAY 20

Working with Different Processors

Yesterday you learned about processor extensions. These extensions are not part of the XSLT language, but enhancements that make some otherwise complicated XSLT tasks easy or even impossible tasks possible. Processor extensions are processor dependent, so in a cross-platform, cross-processor environment, they are useless.

Extensions aren't the only things that make processors different. Because the XSLT specification is open for interpretation, processors can differ somewhat in how certain functionality is implemented. These differences don't mean that your code will work in one processor and not in another, but they have bearing on the resulting output. This lesson looks at some of the differences and how to deal with them (if possible).

Today's lesson also looks at how to use extensions from different processors and how to fall back to other solutions when these extensions are not available. Closely related to this topic is the way you can deal with version differences when new versions of XSLT become available.

Finally, this lesson takes a peek at the XML capabilities of two prominent database servers.

In today's lesson, you will learn the following:

- What the differences are between processors
- How to deal with processor differences
- How to program for different XSLT versions
- What XML capabilities you might see in database servers

Targeting Multiple Processors

One of the main benefits of XML and XSLT should be that you can use them on any platform and with any processor that is compliant with the specification. This situation is true up to a point. XML documents are strings, so they can be used on any platform and with any parser or processor. The glitch is that specifications are not implementations and that specifications are open for interpretation. The result is that not every processor deals the same way with XML documents and stylesheets. So, based on the same source document and the same stylesheet, different processors might create slightly different output. When you create XML output, these differences don't matter much in most cases because the output is semantically the same. The differences manifest themselves in the insertion of whitespace, for instance, but because the specification says that insignificant whitespace might be inserted anywhere, the XML doesn't differ from the specification's point of view. For any subsequent operations on the document, by the original processor or another one, the different documents are treated as equal. The same goes for HTML, where the browser is likely to treat the output as the same HTML as well. The trouble starts in those cases in which this isn't the case.

Another area where processors might differ is in the XSLT version they support and the extensions the processors offer. Elements and functions available in a new version of XSLT might not be available in earlier versions. The reverse is also possible. In a new version, elements and functions from the older version might no longer be available, or its functionality might have changed. The same arguments go for extensions, with the additional problem that extensions are processor specific.

Because processor differences are a fact of life, the only thing you can do is live with them. Fortunately, you can use a mechanism to check whether elements or functions are available, so you can create a workaround if this is the case.

Key Processor Differences

The XSLT specification is such that most elements and functions behave exactly the same in different processors. It would actually be surprising if this weren't the case because that would mean XSLT isn't processor independent, which it is supposed to be. Some elements and operations might yield a slightly different result with different processors, however.

Most processor differences are intricate. These differences do not surface under normal conditions. The only way you can track these differences is through testing your stylesheet with several source documents on the different processors you want to target. As I said earlier, these differences don't matter much for the meaning of the data, but the data might look different. This means that you don't need to work around the differences. It is important, though, that you are aware of the differences so that you're not surprised when something doesn't look exactly as you thought it would.

Whitespace Handling

When creating XML output, a processor might strip or insert insignificant whitespace anywhere it sees fit. Basically, whitespace is inserted into the output as it is matched from the source document, so the whitespace in the output will resemble that in the source document. You can influence this use in two ways. You can use `xsl:strip-space` and `xsl:preserve-space` to tell the processor which insignificant whitespace it can strip from the source document. These two elements behave the same in any processor. If you strip all insignificant whitespace, you are 100% responsible for inserting linefeeds and so on.

The `indent` attribute of the `xsl:output` element has a significant impact on linefeeds and whitespace. If its value is set to `yes`, the processor might indent the code nicely. The glitch is that a processor is not required to indent, and if it does, there is no definition of what the output should look like, so it is likely to differ for different processors. This aspect is important when your documents contain elements with mixed content. Because the processor might insert whitespace to indent the code, this extra whitespace might change the value of a mixed content element because the whitespace inserted is significant. The bottom line here is that two different processors will yield semantically different code, whereas the aim is to keep the documents semantically the same. You can get around this problem in only one way: by setting the value of the `indent` attribute to `no`. If you set the `indent` attribute to `no`, you are responsible for all indentation yourself.

Character Encoding

The character encoding of a source document or stylesheet is of no concern to the processor; this is the domain of the parser that reads in the documents and passes them on to the processor. The character encoding of the output is the responsibility of the processor, however. Any processor must support at least UTF-8 or UTF-16 encoding, but not both. Any other output formats are not required. Basically, this means that if you specify the character encoding with the `encoding` attribute of the `xsl:output` element, you can't be sure that the output will be in the encoding you specified. The XSLT specification describes the `encoding` attribute as the "preferred" encoding. In addition, the specification indicates that a processor might report an error or use the default encoding of that processor instead. It might also do both.

The fact that a processor uses its default encoding instead of the preferred encoding isn't a big deal in most cases, except when you're targeting a platform or application that requires a certain encoding. In that case, you have no choice but to go with a processor that supports the encoding you require. For any processor that doesn't support it, you need to stop processing and generate an error message. To do so, you first need to check which processors produce the correct output. Then, in your stylesheet, you need to take different actions based on the processor vendor. How you do so is discussed in the section "Dealing with Processor Differences" later in this lesson.

Even more annoying is the fact that some processors do not produce any output when they don't support the preferred character encoding. These processors report an error and then stop processing. If you want to be able to show the users anything intelligent, you're out of luck because there's no way to get around this problem, except to use UTF-8 encoding. If you invoke the processor from a program, that program knows which processor it is using, so you could generate an error message from that program.

Conflict Handling

Several elements in a document, such as the `xsl:output` element, might occur one or more times. A problem arises when the values of the attributes are in conflict. For instance, if you use the `xsl:output` element twice, once specifying UTF-16 encoding and once specifying UTF-8 encoding, there is a conflict. The same goes for templates that are redefined and attributes in composite attribute-sets. These cases are more likely to happen when you use `xsl:include` to include other stylesheets. Just as with the preferred encoding, the processor can report an error and stop processing, or it can use the attribute's value in the stylesheet's last element (in document order). If you want to deal with this situation, you have to do the same as with the preferred encoding: Check all processors and tell users which processors they can or can't use.

Attribute Order

From a semantic point of view, the order of the attributes in an element is insignificant. An attribute either is or isn't there. Most processors insert attributes in the order you specify them; however, some processors, such as Xalan, do not. In Xalan's case, the attributes are inserted in reverse order. If you ever need to have a document output attributes in a specific order, you can check which processor is used and create different code for the different processors. How you do so is discussed in the section “Dealing with Processor Differences” later in this lesson.

Namespace Handling

When you work with namespaces, the namespace declarations are copied to the output, unless you specify otherwise with the `exclude-result-prefixes` attribute. Namespaces for which the namespace prefix is defined in the `extension-element-prefixes` attribute are an exception here. Some processors exclude that namespace from the output even if the prefix is not stated in the `exclude-result-prefixes` attribute; others do not. This point is quite important because it introduces into your document a namespace that you don’t want in there. So, if you want to make sure that a namespace declaration doesn’t appear in the output unless needed, you should always use the `exclude-result-prefixes` attribute.

Processors also differ in how they treat the `xsl:namespace-alias` attribute. For instance, if you use an alias for the XSLT namespace because you want to create a stylesheet with XSLT, the output for different processors is different. The `xsl:stylesheet` element with MSXSL looks more or less like this:

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

The preceding element is what you would expect, although this is not required by the XSLT specification, which is why Xalan outputs the following instead:

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

Instead of using the aliased prefix `xsl`, Xalan still uses the original alias `out`. The semantics are still the same, however, so there is no problem. Saxon adds both namespaces, as follows:

```
<out:stylesheet xmlns:out="http://www.w3.org/1999/XSL/Transform"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

This example introduces an additional namespace with the same namespace name, but with a different prefix. Although doing so is fine with the specification, the result is undesirable—not that it really gets in the way, but it’s excess baggage.

Performance Differences

The performance differences between the different processors can be significant, depending on the source data and stylesheet. Although these differences have nothing to do with differences you need to handle, they can be a consideration when you create an application that is meant for a single platform.

You can find benchmarking information and results at `http://www.tfi-technology.com/xml/xslbench.html`. This benchmark consists of three different stylesheets, which are demanding on the processor—so much so, in fact, that some processors fail on some of the tests. This information can be very important if you need to choose between processors. Preliminary tests with the processor that is part of the .NET Framework suggest that this processor blows away the competition, being up to four times as fast as any of the competitors, including Microsoft's own MSXML component. So, if you want to create an application for the Windows platform, the choice is more between conventional technology and the .NET Framework. This will also decide for you which processor to use.

Dealing with Processor Differences

You can deal with processor differences by using different techniques. Each technique is suited for a different case.

Checking the Processor's Properties

Using the `system-property()` function, you can access information regarding the processor vendor and XSLT version that the processor implements. Based on this information, you can choose to do something in one way or another. For instance, you can display a message that the processor doesn't support a particular output encoding or change the order of attributes you insert. The `system-property()` function has one argument, which, by default, can have three values:

- `xsl:version`—This argument returns the XSLT version that the processor implements.
- `xsl:vendor`—This argument returns the processor's vendor. The vendor also might choose to add the product name and version, for example.
- `xsl:vendor-url`—This argument returns a URL to the vendor's Web site.

You might be able to query for additional properties, but they are specific to the vendor, so you might need to check the vendor first. The stylesheet in Listing 20.1 displays the values in the preceding list.

LISTING 20.1 Stylesheet Showing System Properties

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

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
     <xsl:text>Vendor:       </xsl:text>
```

LISTING 20.1 Continued

```
8:       <xsl:value-of select="system-property('xsl:vendor')" />
9:       <xsl:text>&#xA;Vendor URL:   </xsl:text>
10:      <xsl:value-of select="system-property('xsl:vendor-url')" />
11:      <xsl:text>&#xA;XSLT version: </xsl:text>
12:      <xsl:value-of select="system-property('xsl:version')" />
13:   </xsl:template>
14: </xsl:stylesheet>
```

Note You can download the sample listings in this lesson from the publisher's Web site.

ANALYSIS Listing 20.1 displays the values of each system property. The `system-property()` function is used with different argument values on lines 8, 10, and 12. The results for Saxon and MSXSL are shown in Listings 20.2 and 20.3.

OUTPUT **LISTING 20.2** Output from Listing 20.1 for Saxon

```
Vendor:       SAXON 6.2.2 from Michael Kay
Vendor URL:   http://users.iclway.co.uk/mhkay/saxon/index.html
XSLT version: 1
```

OUTPUT **LISTING 20.3** Output from Listing 20.1 for MSXSL

```
Vendor:       Microsoft
Vendor URL:   http://www.microsoft.com
XSLT version: 1
```

You can use the `system-property()` function to conditionally execute (parts of) a stylesheet, as shown in Listing 20.4.

LISTING 20.4 Partial Stylesheet Requiring UTF-16 Encoding

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

  <xsl:output method="text" encoding="UTF-16" />

  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="starts-with(system-property('xsl:vendor'),'SAXON')">
```

LISTING 20.4 Continued

```
9:          <xsl:text>Saxon does not support UTF-16 output.&#xA;</xsl:text>
10:         <xsl:text>UTF-16 output is required.&#xA;</xsl:text>
11:         <xsl:text>Please choose another processor.</xsl:text>
12:       </xsl:when>
13:       <xsl:otherwise>
14:         <xsl:apply-templates />
15:       </xsl:otherwise>
16:     </xsl:choose>
17:   </xsl:template>
18:   <!--more templates here-->
19: </xsl:stylesheet>
```

ANALYSIS The stylesheet in Listing 20.4 requires UTF-16 output encoding. Saxon does not support this type of encoding, so line 8 checks whether the vendor information starts with the string `SAXON`, in which case the template displays the message in Listing 20.5. If you run the code with another processor, the `xsl:apply-templates` element on line 14 is processed instead, so the rest of the stylesheet is executed.

OUTPUT **LISTING 20.5** Output from Listing 20.4 for Saxon

```
Saxon does not support UTF-16 output.
UTF-16 output is required.
Please choose another processor.
```

ANALYSIS When you run Listing 20.4 with Saxon, you get the output in Listing 20.5. In addition, Saxon outputs a message itself on the standard error output. That message shows up only when you run Saxon from the command line. Be aware that other processors that don't support UTF-16 might choose to abort execution altogether, before ever reaching the template matching the root element. Checking for the vendor is therefore more useful in situations in which you need to tweak the output because the output from a different processor is different.

Checking the Existence of Elements

When your stylesheet uses processor-specific extension elements, you can check whether the stylesheet is being run with the right processor by using the method discussed in the preceding section. A more versatile approach is to check whether a certain element exists. If it doesn't, you can take an alternative course of action. You can check for the existence of an element by using the `element-available()` function, which takes one argument, the element you want to check. Listing 20.6 shows some sample data for Listing 20.7. Listing 20.7 shows the `element-available()` function in action.

LISTING 20.6 Sample Data with Numbers

```
<?xml version="1.0" encoding="UTF-8"?>
<numbers>
  <number>2</number>
  <number>9</number>
  <number>144</number>
  <number>65536</number>
  <number>123456789</number>
</numbers>
```

LISTING 20.7 Stylesheet Demonstrating the `element-available()` Function

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4:    xmlns:saxon="http://icl.com/saxon"
5:    extension-element-prefixes="saxon"
6:    exclude-result-prefixes="saxon">
7:
8:    <xsl:output method="text" encoding="UTF-8" />
9:    <xsl:strip-space elements="*" />
10:
11:   <xsl:template match="/">
12:     <xsl:choose>
13:       <xsl:when test="element-available('saxon:assign')">
14:         <xsl:variable name="total" select="0" saxon:assignable="yes" />
15:         <xsl:for-each select="/numbers/number">
16:           <saxon:assign name="total" select="$total - number(.)" />
17:         </xsl:for-each>
18:         <xsl:value-of select="$total" />
19:       </xsl:when>
20:       <xsl:otherwise>
21:         <xsl:text>saxon:assign is not available.</xsl:text>
22:       </xsl:otherwise>
23:     </xsl:choose>
24:   </xsl:template>
25: </xsl:stylesheet>
```

ANALYSIS Listing 20.7 takes Listing 20.6 as input and subtracts all the numbers in Listing 20.6 from 0. To do so, line 15 iterates through all `number` elements in Listing 20.6. Line 16 uses the `saxon:assign` element to alter the value of the variable `total`, which is set as `assignable` on line 14. Before this all happens, line 13 checks whether the `saxon:assign` element is available at all. If it is, the code is executed; otherwise, line 21 is executed. Line 21 just outputs a message that the element is not available, but you can imagine that line 21 could also invoke an alternative calculation method.

Checking the Existence of Functions

In the preceding section, you learned how to check whether an element is available. Besides extension elements, there are also extension functions. Like you do with extension elements, you sometimes need to check whether the function you want to use exists. You can do so by using the `function-available()` function, which works the same as the `element-available()` function. It also can be used to create stylesheets that are not restricted to a single processor but still use extension functions to solve certain problems. The stylesheet in Listing 20.8 uses the `function-available` function in such a situation.

LISTING 20.8 Stylesheet with Extensions Using Different Processors

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:msxsl="urn:schemas-microsoft-com:xslt"
      xmlns:user="http://www.aspnl.com/xmlns/extensions"
      xmlns:math="java.lang.Math"
      extension-element-prefixes="math msxsl user"
      exclude-result-prefixes="math msxsl user">

  <msxsl:script language="javascript" implements-prefix="user">
     function squareroot(num) {
       return Math.sqrt(num);
     }
  </msxsl:script>

  <xsl:output method="text" encoding="utf-8" />

  <xsl:template match="number">
    <xsl:text>The square root of </xsl:text>
    <xsl:value-of select="." />
    <xsl:text> is </xsl:text>
    <xsl:choose>
      <xsl:when test="function-available('math:sqrt')">
        <xsl:value-of select="math:sqrt(.)" />
      </xsl:when>
      <xsl:when test="function-available('user:squareroot')">
        <xsl:value-of select="user:squareroot(number(.))" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>NOT AVAILABLE</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 20.8 calculates the square root of the numbers in Listing 20.6 either using an extension function from a Java class or through a script embedded in the stylesheet. Line 4 is the namespace declaration for MSXSL extensions. It is followed by a namespace for user-defined extensions. Line 6 declares a namespace that defines the Java `Math` class. Lines 7 and 8 define these namespaces as extension prefixes and tell the processor not to copy them to the output. An `msxsl:script` block starts on line 10. In it, a function called `squareroot` is created as part of the `user` namespace. The template on line 18 matches each number in Listing 20.6. For each number, it calculates the square root. Line 23 checks whether the square root can be calculated with the Java function `math:sqrt()`. This function is available in any processor that allows the definition of Java extension functions, as done here; this includes Saxon, Oracle XSL, and Xalan. If the function isn't available, line 26 checks whether a user-defined function is available. In this case, this function is defined only for MSXSL, so if the processor you use is MSXSL, line 27 calculates the square root. If both functions aren't available, line 30 tells the users so. Listing 20.9 shows the output from Listing 20.8 when applied to Listing 20.6.

OUTPUT **LISTING 20.9** Result from Applying Listing 20.8 to Listing 20.6

```
The square root of 2 is 1.4142135623730951
The square root of 9 is 3
The square root of 144 is 12
The square root of 65536 is 256
The square root of 123456789 is 11111.111060555555
```

Listing 20.9 will show the square root calculations only if you run it from a processor that supports Java extensions or from MSXML. Other processors will yield the text `NOT AVAILABLE`. This still means, however, that most common processors will produce the result shown in Listing 20.9.

Dealing with Different XSLT Versions

In future versions of XSLT, new elements might be introduced. This no doubt means that at one point or another there will be processors for the new version but also older versions that don't support these elements yet. The problem that then arises is that the processor doesn't recognize the new element and fails. You can check whether a processor recognizes these elements first by using the `element-available()` function, but there is a more graceful alternative. You can use the `xsl:fallback` element to give an alternative course of action for the parent element. This element might contain any elements that are not top-level elements, so you can match or call other templates, use different functions, and so on. Listing 20.10 shows `xsl:fallback` in action.

LISTING 20.10 Stylesheet Using Fallback to a Previous XSLT Version

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.5"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:
 5:   <xsl:output method="text" encoding="UTF-8" />
 6:
 7:   <xsl:template match="/">
 8:     <xsl:sum select="/numbers/number">
 9:       <xsl:fallback>
10:         <xsl:value-of select="sum(/numbers/number)" />
11:       </xsl:fallback>
12:     </xsl:sum>
13:   </xsl:template>
14: </xsl:stylesheet>
```

Note

The stylesheet in Listing 20.10 uses an XSLT version that is *not* correct. The XSLT version used here doesn't exist and probably never will.

ANALYSIS The stylesheet in Listing 20.10 was created with the imaginary XSLT version 1.5, as you can see from line 2. For the sake of this example, XSLT version 1.5 no longer supports the `sum()` function but uses the `xsl:sum` element on line 8 instead. This element is clearly not available in XSLT 1.0. In the `xsl:sum` element's body, an `xsl:fallback` element is added on line 9, and this element reverts back to the `sum()` function on line 10, supported by the current version. The idea is that when you process this stylesheet with a processor that supports any version before version 1.5, the processor checks whether it has the `xsl:sum` element available. If it does not, it reverts back to the functionality inside the `xsl:fallback` element, which it assumes is supported by the processor.

Caution

The `xsl:fallback` element relies heavily on the version number. If the version on line 2 in Listing 20.10 had been 1.0, the processor would have generated an error because the element is not supported by version 1.0. When the 1.0 processor encounters the version 1.5 definition, it assumes that there might be elements it can't handle.

You also can use the `xsl:fallback` element in situations in which you're using extension functions. When the stylesheet is run by a processor that's different from the processor you use, the `xsl:fallback` element can give an alternative, as shown in Listing 20.11.

LISTING 20.11 Stylesheet Using Fallback for Extension Elements

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xsl:stylesheet version="1.0"
 3:   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4:   xmlns:saxon="http://icl.com/saxon"
 5:   extension-element-prefixes="saxon"
 6:   exclude-result-prefixes="saxon">
 7:
 8:   <xsl:output method="text" encoding="UTF-8" />
 9:   <xsl:strip-space elements="*" />
10:
11:    <xsl:template match="/">
12:     <xsl:variable name="total" select="0" saxon:assignable="yes" />
13:     <xsl:for-each select="/numbers/number">
14:       <saxon:assign name="total">
15:         <xsl:fallback>
16:           <xsl:text>saxon:assign is not available.</xsl:text>
17:         </xsl:fallback>
18:         <xsl:value-of select="$total - number(.)" />
19:       </saxon:assign>
20:     </xsl:for-each>
21:     <xsl:value-of select="$total" />
22:   </xsl:template>
23: </xsl:stylesheet>
```

ANALYSIS Listing 20.11 uses the `saxon:assign` element to total the numbers in Listing 20.6. Instead of checking for its existence with the `element-available()` element, the element is just used on line 14, but with an `xsl:fallback` element on line 17 to guard against processors other than Saxon. Note that there is no check for the `saxon:assignable` attribute. Because it is an attribute and not in the `xsl` namespace, that check isn't necessary. Note that the `saxon:assign` element now doesn't use the `select` attribute to assign the new values, but the value is given by line 18 instead. Because the `saxon:assign` element has a body, the `select` attribute isn't allowed, just like when you create a variable with the `xsl:variable` element.

XML Capabilities of Database Servers

XML is very much seen as the data format of the future, and database vendors are quickly providing methods to get XML from a database and, if needed, transform the result to a readable format for different target platforms. Databases that can return XML, instead of legacy data formats, are an important step forward. Text documents have their place, but a problem arises when several people have to retrieve and alter parts of the data in a document simultaneously. Databases are designed to provide users with such capabilities, circumventing problems associated with concurrent access of a data source.

In the following sections, I will discuss the XML capabilities of two of the most dominant database servers: Oracle and Microsoft SQL Server. Both enable the user to send a query to the database through the Web server. The result is sent back to the client as XML. Because the client performs the query through the Web server, the data is potentially available from anywhere on the Internet.

Getting and Transforming XML Data with Oracle

Since Oracle 9i, the XML capabilities have really taken a solid shape. The basic components you need to work with XML from an Oracle database are the Oracle 9i (or higher) database itself, the J2EE platform, and a Web server that can work with Oracle, such as Apache. The Oracle 9i distribution, by default, delivers all these components. Also, part of Oracle is the XSQL Servlet, which is a component that drives the capability to query Oracle for XML data. The XSQL Servlet needs to be installed on the Web server to enable you to query Oracle for XML.

Note

Installing and configuring Oracle and the XSQL Servlet are beyond the scope of this book. You can find more information at `http://otn.oracle.com/tech/xml`.

New Term When all the components are installed and configured properly, you can create documents for the XML query process in a *virtual directory* on the Web server. A virtual directory is accessible through the Web. You query for XML with an XSQL document, which is a special XML document that contains a database query in Structured Query Language (SQL).

To show you what XSQL and the XSQL Servlet do, I've created an example that assumes there is a table in an Oracle database that stores information on cars. This table, named CarModels, contains three fields: Model, Manufacturer, and Year. The information in this table is similar to the car data used in examples from previous lessons. Table 20.1 shows the data in the database table.

Table 20.1 Table with Car Information

Model	*Manufacturer*	*Year*
Golf	Volkswagen	1999
Camry	Toyota	1999
Focus	Ford	2000
Civic	Honda	2000
Prizm	Chevrolet	2000

TABLE 20.1 Continued

Model	*Manufacturer*	*Year*
Celica	Toyota	2000
Mustang	Ford	2001
Passat	Volkswagen	2001
Accord	Honda	2002
Corvette	Chevrolet	2002

To enable users to query for data, you need to create an XSQL document in a virtual directory on the Web server. This document contains a predefined query, or multiple queries, that the user can invoke by requesting the XSQL document. In the document, you can define parameters that the user can give to obtain a different result. Listing 20.12 is an example of an XSQL document that performs a query on the data in the CarModels table.

LISTING 20.12 XSQL Document with a Query for Car Models

```
1: <?xml version="1.0"?>
2: <xsql:query connection="cardatabase"
3:       bind-params="Manufacturer" xmlns:xsql="urn:oracle-xsql">
4:   SELECT Model, Manufacturer, Year
5:   FROM CarModels
6:   WHERE Manufacturer = ?
7:   ORDER BY Model
8: </xsql:query>
```

ANALYSIS Listing 20.12 contains an XSQL document, which, as you can see, is an XML document. The document starts with a proper XML prolog and then defines an `xsql:query` element on line 2. This element contains the SQL query that needs to query the database when a user requests the XSQL document. The `connection` attribute of the `xsql:query` element tells the XSQL Servlet which database to use—in this case, `cardatabase`. The properties of this connection are defined in a separate XML filename XSQLConfig.xml. On line 3, the `bind-params` attribute defines the parameters that a user can pass to get certain data from the query. For this query, the user can select car models based on a manufacturer. The `xsql` namespace is also defined on line 3. Note that the namespace name is a URN specific to Oracle. The `xsql:query` element contains a normal SQL query. Line 4 selects the Model, Manufacturer, and Year fields, with line 5 telling the database that these fields need to be selected from the CarModels table. Line 6 narrows the selection to a certain manufacturer. The question mark takes the place of the parameter defined in the `bind-params` attribute on line 3. When the user makes the request, specifying a certain manufacturer, the question mark is replaced by that manufacturer. Finally, line 7 makes sure that the models returned are ordered alphabetically on Model.

If the XSQL document in Listing 20.12 is saved to the root directory of the Web server as `CarModels.xsql`, a user can request it by using the following URL:

```
http://www.example.com/CarModels.xsql?Manufacturer=Ford
```

After the question mark, the user can specify the manufacturer to get the car models for, in this case, Ford. When this request is made, the XSQL Servlet replaces the question mark on line 6 of Listing 20.12 with Ford before querying the database. The data returned from the database is then converted to an XML file like the one shown in Listing 20.13.

OUTPUT

LISTING 20.13 Result from Performing a Query with the XSQL Document in Listing 20.12

```
1:  <?xml version="1.0"?>
2:  <ROWSET>
3:    <ROW num="1">
4:      <MANUFACTURER>Ford</MANUFACTURER>
5:      <MODEL>Focus</MODEL>
6:      <YEAR>2000</YEAR>
7:    </ROW>
8:    <ROW num="2">
9:      <MANUFACTURER>Ford</MANUFACTURER>
10:     <MODEL>Mustang</MODEL>
11:     <YEAR>2001</YEAR>
12:   </ROW>
13: </ROWSET>
```

ANALYSIS

In Listing 20.13, each record retrieved from the database is designated by a `ROW` element, which has a `num` attribute, with a unique number value for each record. As you can see, the whole result is embedded in a `ROWSET` element, which starts on line 2. Each field in the records is returned as an element with the field's name. The value of this element is the value in the database. Lines 4 and 9 both contain a `MANUFACTURER` element. Because the request was only for Ford, that is the value of both.

If you perform the request resulting in Listing 20.13 with a client that can process XML, you have no problem. When you request it from a client that doesn't process XML, such as a Web browser, you want the data to look nice. You can make it look good by attaching stylesheets to the XSQL document. These stylesheets are then used on the Web server to transform the XML result from the query to other formats, such as HTML or WML.

The result from the transformation is then sent to the client. Listing 20.14 shows the XSQL document from Listing 20.12, but with several stylesheets attached to it.

LISTING 20.14 Listing 20.12 with Several Stylesheets Attached

```
1:  <?xml version="1.0"?>
2:  <?xml-stylesheet type="text/xsl" media="MSIE 5.0" href="IE5HTML.xsl"?>
3:  <?xml-stylesheet type="text/xsl" media="HandHTTP" href="Palm.xsl"?>
4:  <?xml-stylesheet type="text/xsl" href="HTML.xsl"?>
5:  <xsql:query connection="cardatabase"
6:        bind-params="Manufacturer" xmlns:xsql="urn:oracle-xsql">
7:    SELECT Model, Manufacturer, Year
8:    FROM CarModels
9:    WHERE Manufacturer = ?
10:   ORDER Model
11: </xsql:query>
```

ANALYSIS Lines 2–4 define different stylesheets for different target clients. Line 4 defines the default stylesheet, which is used if none of the other stylesheets apply. The other stylesheet definitions have a `media` attribute, which tells the XSQL Servlet when to use which stylesheet. The stylesheet on line 2 should be used only for Internet Explorer 5, and the stylesheet on line 3 only for hand-held devices. The Web server detects the type of client used to access the data and will apply the proper stylesheet. A graphical representation of this process is shown in Figure 20.1.

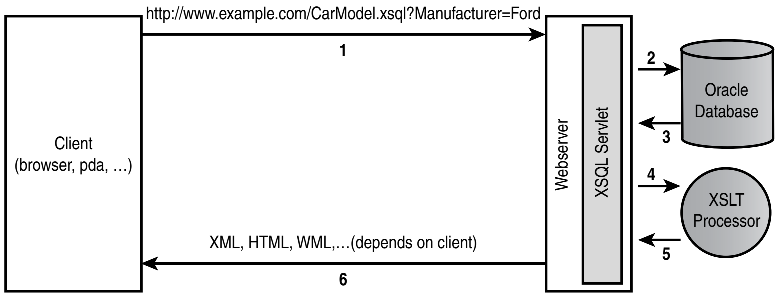

http://www.example.com/CarModel.xsql?Manufacturer=Ford
1
Client
(browser, pda, ...)
Webserver
XSQL Servlet
2
3
Oracle
Database
4
XSLT
Processor
5
XML, HTML, WML,...(depends on client)
6

FIGURE 20.1
Overview of an XSQL Servlet request.

Figure 20.1 shows the steps involved for a client to get a result from the Oracle database through the XSQL Servlet. These steps are as follows:

1. The client requests an XSQL document from the Web server, which then invokes the XSQL Servlet to deal with the XSQL document. The request could contain one or more variables, but this is not required.
2. The XSQL Servlet takes the SQL statement from the XSQL document and queries the database.

3. The database returns a result. The XSQL Servlet makes an XML document for that result.
4. The XSQL Servlet invokes the XSLT processor to transform the XML given the stylesheet attached to the XSQL document for the client type making the request.
5. The XSLT processor returns the result, which can be plain text, XML, HTML, WML, or another format.
6. The XSQL Servlet returns the result to the Web server, which returns it to the client.

If no stylesheets are specified, steps 4 and 5 are skipped and the XML result is returned to the client immediately.

Note

You can do much more with the XSQL Servlet, but discussing it any further is beyond the scope of this book. You can find more information at `http://download-eu.oracle.com/otndoc/oracle9i/901_doc/appdev.901/a88894/adx10xsq.htm`.

Getting XML Data from Microsoft SQL Server

As you learned in the preceding section, Oracle offers one unified system to query for XML and transforming it, if necessary. In SQL Server, support for retrieving XML data from a database is much more elaborate. On the other hand, SQL Server doesn't support XSLT transformations on the server side by itself. Microsoft provides a separate component named XSL ISAPI to do transformation on the server side, which needs to be integrated programmatically with the SQL Server XML query facilities. One reason for this lack of integration is that Internet Explorer now supports client-side transformation, and Microsoft obviously thinks that this will be true for more client types in the future. This means that, at this point, most of what you would want to achieve in functionality similar to Oracle's XSQL Servlet needs to be programmed outside XML and XSLT, which is not what you, as an XML/XSLT programmer, want to hear. The other reason is that a database server is not meant to serve data directly to a user, but rather through another tier in the system that formats the data. So, the XML support of SQL Server is aimed more at tier-to-tier communication than client-communication. The main reason for doing this type of communication in XML is that the two tiers can communicate over the Web.

SQL Server basically offers two methods of getting XML data. One is through a SQL query that you can send to the server as part of the URL. Microsoft has created an addition to the SQL language to specify that you want to get the data back as XML. A URL that selects data in this manner is shown in Listing 20.15.

LISTING 20.15 Sample URL for Selecting Data from SQL Server

```
http://www.example.com/cardatabase?template=<ROWSET+
➥xmlns:sql="urn:schemas-microsoft-com:xml-sql">
➥<sql:query>SELECT+Model,Manufacturer,Year+FROM+CarModels+WHERE+
➥manufacturer+='Ford'+FOR+XML+RAW</sql:query></ROWSET>
```

ANALYSIS Listing 20.15, when typed in a browser, will return a document similar to the result received in Listing 20.13. The outside element is explicitly created as a `ROWSET` element to mimic the Oracle syntax. The `FOR XML RAW` syntax at the end of the SQL query ensures that the data inside the `ROWSET` element is indeed XML. The actual result from Listing 20.15 is shown in Listing 20.16.

OUTPUT **LISTING 20.16** Result from Performing a Query with the URL in Listing 20.15

```
<ROWSET xmlns:sql="urn:schemas-microsoft-com:xml-sql">
   <row Model="Focus" Manufacturer="Ford" Year="2001" />
   <row Model="Mustang" Manufacturer="Ford" Year="2002" />
</ROWSET>
```

ANALYSIS Listing 20.16 doesn't differ that much from Listing 20.13. Here, the rows returned from the query are represented as elements, with the fields as attributes. In Listing 20.13, the fields themselves are also elements.

Note

For a more complex method, SQL Server offers to query for XML through annotated XML Schemas that reside on the server, more or less like the XSQL documents in Oracle. Using this method, you have much more control over the structure of the resulting XML. Discussing this method is beyond the scope of this book because it requires extensive knowledge of XML Schemas.

Summary

Today you learned that different processors have tiny differences in their output because the XSLT specification leaves room for interpretation. In most cases, this isn't such a problem, but you need to guard yourself against the cases in which it is. The major differences lie in supported character encoding, whitespace handling, and conflict resolution with multiple elements, such as the `xsl:output` element. These differences can have a profound impact when a processor stops processing because of such a situation. Other processor differences lie in the support of extension functions and elements and in the XSLT version that future processors might support.

You can get around many of the processor differences by checking availability of functions and elements with the `function-available()` and `element-available()` functions. More graceful for element support is using the `xsl:fallback` element as an immediate child to give an alternative course of action. This method is mainly meant for dealing with different XSLT versions, but it can be applied to extension elements as well.

In tomorrow's lesson, you will learn more details about how to design and set up XML and XSLT documents. This information will better your understanding of XML and XSLT as a whole and will help you create better applications.

Q&A

Q What are the chances that all processors will behave exactly the same in future implementations of XSLT?

A Very small. First, a specification is almost always open for interpretation. In addition, the difference between processors aimed at the Java platform and Microsoft platform will probably not change in the near future.

Q Can I use `xsl:fallback` with processor extensions from different processors in the same stylesheet?

A Yes. You can even leave the `xsl:fallback` element empty in each case so that nothing happens. Only the supported element will be run properly, so you have a more or less processor-independent stylesheet although you're using extension elements.

Q When will I have to worry about different XSLT versions?

A A new version of XSLT must first reach W3C Recommendation status before it is really accepted as a new version. At the time of this writing, XSLT version 1.1 is a Working Draft, which is one step before becoming a Recommendation. Because work on XSLT version 1.1 has been abandoned in favor of XSLT version 2.0, it is not likely that version 1.1 will ever become a Recommendation. Version 1.1 does not differ very much, however, except for the `xsl:script` element and some minor details. XSLT version 2.0 is still on the drawing board. It has yet to reach Working Draft status, so it will take some time before it becomes a Recommendation.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: The `xsl:fallback` element can be used as an alternative for the `function-available()` function.
2. True or False: The `xsl:fallback` element can be used only to deal with different XSLT versions.
3. Why can `indent="yes"` as part of the `xsl:output` element be responsible for changing the semantics of a document?
4. What is the use of the `system-property()` function?
5. Does SQL Server support XSLT transformation before returning a query as XML?

Exercise

1. Using XSLT version 1.0, you cannot send output to multiple documents. Several processors provide extension elements for this functionality, until XSLT supports its own mechanism. Saxon uses the `saxon:output` element, which has an `href` attribute indicating the file to be created. From the following source XML, create a stylesheet that creates an HTML file for each news item, with the normal output stream creating an index HTML file that links to each document. Use `xsl:fallback` and the `element-available()` function to create one HTML document with all the news items for processors other than Saxon.

```
<?xml version="1.0" encoding="UTF-8"?>
<news>
  <newsitem id="2" title="New World Record 100m" date="09/12/2040">
    In a phenomenal phenomenal race against rivals Lewis Carlson and
    Chris Linford, Canadian sprinter John Benson ran the 100 meters
    in a new world record time of 4.98 seconds.
  </newsitem>
  <newsitem id="3" title="Michael Jordan Makes Comeback" date="09/14/2040">
    At age 77 Michael Jordan is the oldest player to make a comeback in the
    NBA. After a year in a retirement home in Florida he decided he could
    still teach the young stars a thing or two.
  </newsitem>
  <newsitem id="4" title="Moon Race Cancelled" date="09/20/2040">
    Due to increased solar activity this year's Armstrong Cup has been
    cancelled. The organization can't guarantee the safety of the
    participants in the race around the moon.
  </newsitem>
</news>
```

This page intentionally left blank

DAY 21

Designing XML and XSLT Applications

In the preceding lessons, you learned about all the individual pieces of the XSLT language. On Day 19, "Working with XSLT Extensions," you even learned how to extend the XSLT language, and in yesterday's lesson, you learned how to deal with the differences between processors.

Today you will learn more about the bigger picture when creating XML and XSLT applications. Instead of discussing individual elements or functions, this lesson looks at design considerations when you're creating XML and XSLT applications. This does not mean that this lesson will give you the ultimate way to design applications. The discussion centers around the questions that will face you each time you have to create an application and how to answer these questions on an application-by-application basis.

In today's lesson, you will learn the following:

- How to decide whether your XML should contain elements or attributes for certain values
- Which options you have when defining the hierarchy of your XML

- How to decide between matching and selection
- When to use variables or keys

Designing XML

XML is a versatile data storage format. For most data, you have a multitude of options on how to set up the data structure. The choices you make will have a lasting effect on your application, so you should design your XML structures with care.

Important aspects to the design of your XML structures for an application are the application domain and use of data. If the data will be displayed without much change to the data itself, using a format that closely resembles the formatted output makes creating your stylesheets much easier because you don't have to shuffle data around. On the other hand, if the data needs to be processed and altered before being displayed, you will want to create a format that is most suited for manipulating the data, without resorting to complex expressions or variables.

Closely linked to the application domain is the way XML is used in an application. Using it for storage and display is one thing, but it might also be used to transmit data between sections of an application. For instance, if your application is an order system, data from an order must be passed on to invoicing, shipping, and so on. Before you start to work on the XML design itself, you should have a design of the application itself—for example, in Unified Modeling Language (UML). Specifically, sequence diagrams that show what happens on certain actions and state diagrams that show you the state of the application under certain conditions are useful. Such diagrams show you where XML might be needed and which data should be stored in XML or transmitted with XML.

The problem with design is that deciding which options you should go for is not always clear-cut. One reason you store data in an XML document in the first place is that you can create different stylesheets to create different output. This means that if you design the structure of your XML document from the point of view of one of the output formats, creating a stylesheet for the other format might be a tall order. You therefore need to consider what the common denominators are and design from there. This is a pure methodological breakdown of your application. You can start by creating sample output for each output you need to create from an XML source. Chances are you'll see structures that are similar in each case. Those structures are likely to have a similar structure in your XML structure.

One important point to remember is that you can restructure your XML documents. So, you can create a first draft of your design that holds all the (sample) data. While you're creating the draft, you will likely see that some things work well and some things don't. From the draft, you can create sections of the different output that you'll need for your application. This way, you can quickly see where you will encounter problems if you keep the structure as it is. As long as you stick to templates as much as possible, changes to the data don't affect your stylesheets very much. In many cases, you can alter a template just slightly so that it will create the same output but from data that is structured differently.

Consider the following structure:

```
<name firstname="Michiel" lastname="van Otegem" />
```

A template processing this structure might look like this:

```
<xsl:template match="name">
  <xsl:value-of select="@lastname" />
  <xsl:text>, </xsl:text>
  <xsl:value-of select="@firstname" />
</xsl:template>
```

Later, you discover that people can have more than one first name, so you change the XML to

```
<name>
  <firstname>Michiel</firstname>
  <lastname>van Otegem</lastname>
</name>
```

Changing the template accordingly is not a big problem; you just need to remove the attribute markers from the data selection so that it'll look as follows:

```
<xsl:template match="name">
  <xsl:value-of select="lastname" />
  <xsl:text>, </xsl:text>
  <xsl:value-of select="firstname[1]" />
</xsl:template>
```

Because of the way XSLT selects data, this template produces the same output as the former, with the corrected XML source. Even if the changes to the template aren't trivial, the other templates in the stylesheet will stay the same, for the most part, because each template acts as an independent unit. Once a unit works well and the XML structure for that unit is final, you can use it indefinitely, even across applications.

During the design phase, you can look at XML as moldable. The data can take the shape you want it to, and if it doesn't work, you can reshape it. You can do this directly with an XML document itself, which is unlike most other data storage formats. For instance, you cannot create and change database tables and relationships quickly. Database design is therefore often done on paper or with design tools. Before you take any action on the database itself, the design must be ready. This is not so for XML, which you can change easily, even during the implementation phase if you encounter a problem. That doesn't mean, of course, that it isn't a good idea to use data modeling tools and techniques.

They have been around for years and are a product of experience. That said, most design techniques and tools are still aimed at the more rigid data formats around, so in the end, you can perform tasks with XML that go beyond what those tools and techniques allow you to do. So, although they are a good starting point, using XML itself during the design phase can help you iron out implementation-type problems before you actually start implementing an application. In fact, the design and implementation phases with XML applications are much closer to each other because the data design is the same as the data format, whereas a database design is nowhere near the actual format.

XML Design Considerations

When you design XML, you always need to make some key considerations. The hierarchy you use is, of course, very important, but also the choice between elements and attributes, and so on. Although there is no definitive answer to the question "What is better?" the following sections will help you decide.

Setting Up a Hierarchy

The hierarchy you use in a document is very important, specifically how you select data using a stylesheet. If two (or more) sets of data are related to one another, how do you define their relationship? One way is to have values that reference each other, as shown in Listing 21.1.

LISTING 21.1 XML Document with Car-Manufacturer Relationship

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <models>
    <model name="Golf" manufacturer="VW" year="1999" />
    <model name="Camry" manufacturer="TY" year="1999" />
    <model name="Focus" manufacturer="FO" year="2000" />
    <model name="Civic" manufacturer="HO" year="2000" />
    <model name="Prizm" manufacturer="CV" year="2000" />
    <model name="Celica" manufacturer="TY" year="2000" />
    <model name="Mustang" manufacturer="FO" year="2001" />
    <model name="Passat" manufacturer="VW" year="2001" />
    <model name="Accord" manufacturer="HO" year="2002" />
    <model name="Corvette" manufacturer="CV" year="2002" />
  </models>
  <manufacturers>
    <manufacturer id="VW" name="Volkswagen" country="Germany" />
    <manufacturer id="TY" name="Toyota" country="Japan" />
    <manufacturer id="FO" name="Ford" country="USA" />
    <manufacturer id="CV" name="Chevrolet" country="USA" />
    <manufacturer id="HO" name="Honda" country="Japan" />
  </manufacturers>
</cars>
```

ANALYSIS In Listing 21.1, the cars and manufacturers are related. Their relationship is defined by the `manufacturer` attribute of the `model` elements. That attribute's value corresponds to the `id` attribute of the `manufacturer` elements. When you want to select data concerning a car's manufacturer, you need to use a predicate expression such as

```
/cars/manufacturers/manufacturer[@id = current()/@manufacturer]/@name
```

The other way around you are faced with the same problem. The preceding expression is less than delightful. Not only does it use a predicate expression to get to the data, but it also relies on absolute addressing to get to the data. If you were to make the data in Listing 21.1 part of a larger structure, the absolute addressing might change, making the expression useless. If you want to make a list with manufacturers and their cars, your stylesheet would look like Listing 21.2.

LISTING 21.2 Stylesheet Creating a List of Manufacturers and Cars from Listing 21.1

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <xsl:stylesheet version="1.0"
3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4:
5:    <xsl:output method="text" encoding="UTF-8" />
6:
7:    <xsl:template match="/">
8:      <xsl:apply-templates select="/cars/manufacturers" />
9:    </xsl:template>
10:
11:   <xsl:template match="manufacturers">
12:     <xsl:for-each select="manufacturer">
13:       <xsl:value-of select="@name" />
14:       <xsl:value-of select="concat(' (',@country,')&#xA;')" />
15:       <xsl:for-each
16:            select="/cars/models/model[@manufacturer = current()/@id]">
17:         <xsl:value-of select="concat('-',@name,' (',@year,')&#xA;')" />
18:       </xsl:for-each>
19:       <xsl:text>&#xA;</xsl:text>
20:     </xsl:for-each>
21:   </xsl:template>
22: </xsl:stylesheet>
```

ANALYSIS Listing 21.2 creates a simple list of manufacturers and cars. To let the processor start processing the manufacturers instead of the cars, the `xsl:apply-templates` element on line 8 selects that data for matching. If that element were omitted, the processor would match the `models` and `model` elements without doing something with the data, which just costs processing cycles. The template on line 11 matches the `manufacturers` element and does all the processing using iteration. This template can hardly do without iteration, especially because of line 16, which selects the cars to be iterated. This expression used with a match template would require additional selection of the elements before invoking other templates as well. The result is as shown in Listing 21.3

OUTPUT **LISTING 21.3** Result from Applying Listing 21.2 to Listing 21.1

```
Volkswagen (Germany)
-Golf (1999)
-Passat (2001)

Toyota (Japan)
-Camry (1999)
-Celica (2000)

Ford (USA)
-Focus (2000)
-Mustang (2001)

Chevrolet (USA)
-Prizm (2000)
-Corvette (2002)

Honda (Japan)
-Civic (2000)
-Accord (2002)
```

Now consider Listing 21.4, which holds the same data as Listing 21.1, but structured differently.

LISTING 21.4 XML Document with Car-Manufacturer Relationship in a Hierarchy

```
<?xml version="1.0" encoding="UTF-8"?>
<manufacturers>
  <manufacturer id="VW" name="Volkswagen" country="Germany">
    <model name="Golf" manufacturer="VW" year="1999" />
    <model name="Passat" manufacturer="VW" year="2001" />
  </manufacturer>
  <manufacturer id="TY" name="Toyota" country="Japan">
    <model name="Camry" manufacturer="TY" year="1999" />
    <model name="Celica" manufacturer="TY" year="2000" />
  </manufacturer>
  <manufacturer id="FO" name="Ford" country="USA">
    <model name="Focus" manufacturer="FO" year="2000" />
    <model name="Mustang" manufacturer="FO" year="2001" />
  </manufacturer>
  <manufacturer id="CV" name="Chevrolet" country="USA">
    <model name="Prizm" manufacturer="CV" year="2000" />
```

Listing 21.4 Continued

```
    <model name="Corvette" manufacturer="CV" year="2002" />
  </manufacturer>
  <manufacturer id="HO" name="Honda" country="Japan">
    <model name="Civic" manufacturer="HO" year="2000" />
    <model name="Accord" manufacturer="HO" year="2002" />
  </manufacturer>
</manufacturers>
```

Analysis Instead of using reference values, Listing 21.3 uses the hierarchy of the XML document to define the relationship of the manufacturers and cars. This makes addressing one from the other easy because they have a relationship that can be defined with relative addressing. If you want the manufacturer of the current `model` element, you can get that element by using the expression `parent::manufacturer`, which is much more friendly than something with reference values and predicates. Listing 21.5 shows how using this hierarchy changes Listing 21.2.

Listing 21.5 Stylesheet Creating a List of Manufacturers and Cars from Listing 21.4

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

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

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

  <xsl:template match="manufacturer">
    <xsl:value-of select="@name" />
    <xsl:value-of select="concat(' (',@country,')&#xA;')" />
    <xsl:apply-templates />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="model">
    <xsl:value-of select="concat('-',@name,' (',@year,')&#xA;')" />
  </xsl:template>
</xsl:stylesheet>
```

ANALYSIS Listing 21.5 creates the same output as shown in Listing 21.3 when it is applied to Listing 21.4. The structure of this stylesheet is different from Listing 21.2, however. Line 8 no longer selects the elements that need to be processed because that is not necessary with the hierarchy in Listing 21.4. Instead of using `xsl:for-each` to iterate through the `manufacturer` elements, the template on line 11 now uses matching. The template on line 15 matches the `manufacturer` elements and outputs their values. Instead of a nested `xsl:for-each` loop, line 18 uses matching again to match the child elements of the `manufacturer` elements. This approach is much easier than the predicate expression used earlier, and in essence relative addressing is used. The template on line 22 outputs the values for the car on line 23. This stylesheet is desirable over the stylesheet in Listing 21.2 because it is much simpler, and each template is a unit of processing that can easily be replaced if you want to create different output. The templates in Listing 21.5 are independent of each other, whereas Listing 21.2 has one bulky template, which is also harder to understand.

As you can see, the difference between Listing 21.1 and Listing 21.4 has a huge impact on how the data can be processed. Most data selections in Listing 21.4 are much easier to accomplish. Also, getting only the `model` elements from Listing 21.4 isn't much harder than getting them from Listing 21.1; using `//model` or `/manufacturers/manufacturer/model` will do the trick.

If you have experience with databases, Listing 21.1 is the obvious choice because you are used to tables with related data. Listing 21.4 is not something you might have come up with because the relationships are hierarchical. XML is, in essence, a hierarchical data format, so hierarchic relationships have preference over other types of relationships. Selecting data is much easier and will probably also perform better, especially with large datasets.

Elements or Attributes?

The debate whether you should use elements or attributes is as old as XML itself. Some people think you shouldn't use attributes at all, only elements. Their argument is that an attribute is just an element that might occur only once and might have only a text value. You can enforce both these qualities with a DTD or a Schema, so why bother with a different notation, which affects XML, DOM, and XSLT? One answer is, of course, that you can enforce these properties with an attribute without having to define a DTD or Schema. There are, however, other considerations between elements and attributes as well.

Attributes take less space in a document than an element because an element needs to have an opening and a closing tag, whereas an attribute needs only quotation marks and the = character. When a document is large, using an attribute can save quite a large amount of space. In a networking environment where bandwidth is a factor, shaving off 20% of a document's size might be very important, specifically if the document needs to be sent over the wire many times—for instance, in a Web-based scenario. In such a scenario, you have to pay per gigabyte of data sent, so a document that is 20% smaller means a savings of 20% on cost.

From a design point of view, there are two important differences between elements and attributes. An element can occur more than once as a child element of another element. In table-like data, as shown in the preceding sections, this is very important, but it also holds true in less structured or hierarchical data structures, such as a `name` element with several `firstname` child elements. Attributes, on the other hand, can occur only once. Having two attributes with the same name is not allowed. So, in some cases, you can store data only as an element. Also, an element is extensible. An attribute can contain only a number or string value, but an element can be extended with additional child elements. This concept is very important because if you choose an attribute and create all stylesheets accordingly, extending the data structure is a tough job. If you use an element, on the other hand, you can add child elements if the need arises.

In essence, attributes are a good choice when you're sure that you can have only one of them, and you will never need to extend them. Values, such as unique identifiers, qualify very well for attributes.

Caution

If you choose to use elements, you need to be aware of side effects that occur when you select the value of a node-set or when no matching template exists. In that case, the element value is written to the output, which is probably not what you intended.

One Document or Multiple Documents?

Whether you should use one or multiple documents to store data is not any easy question to answer. There are several considerations:

- Can the data be structured so it can be divided into several documents?
- How large is the entire dataset?
- How often does the data change?
- Is breaking up the data into smaller pieces testing (parts of) the application easier?
- What is the impact of multiple documents on the complexity of the expressions?

If the data is hierarchical in nature, dividing it into several files is not easy. Listing 21.4 is very hard to break up because you need each manufacturer and the related data to be able to process the document properly. Dividing it into several pieces is not possible, apart from creating a different file for each car model for a manufacturer. Dividing the data that way would almost certainly make your application hard to manage, so that approach is not an option, unless the number of car models per manufacturer is large. If, on the other hand, the data is structured as in Listing 21.1, you can easily divide it into two separate files, one with the manufacturers and one with the cars. When you load one or the other into a variable in the stylesheet, you can access the data in the variable by using the reference values in the other document. The downside is, of course, that this will make your expressions more complex because you have to use predicates to get to the right data.

If the dataset is very large, and you don't always need all the data, breaking up the data into several files is a good idea. The less data the processor has to sift through, the better the performance of your application. In addition, it is probably much easier to test and debug a section of your application with a subset of the data. If you can make sure that all separate sections of your application are correct, you can limit a search for errors in the entire application to the code that is required to use the sections as one large application. This will undoubtedly save you a lot of time.

NEW TERM Another consideration here is how many users need concurrent access to the data. Unless you're working with some kind of database, concurrent access to the data is tricky at best, especially if the data needs to change on a regular basis. In that case, smaller files might help because it is more likely that you can open a file to make changes. You need to be aware if you have files with related data, however; if you change one file, for example, the other might not have the referenced data until you change that file as well. With most stylesheets, this is not a problem because, unlike a database, a stylesheets doesn't enforce *referential integrity*, which means that data might exist without the reference existing.

One point you need to keep in mind when working with multiple documents is that if you use matching on a secondary document loaded with the `document()` function, the data in the original source document is not available unless it, too, is stored in a variable. This also goes for keys defined on the source document. The keys you have defined work only while matching the source document for which they were defined, so cross-document keys are not possible. In essence, the more documents you have, the harder it becomes to use data from those documents in concert.

Using Namespaces

In a simple, small application, namespaces are often more trouble than they are worth. If, however, you create an application consisting of different datasets with disjoint vocabularies, possibly consisting of multiple documents, namespaces are an absolute must. Namespaces make sure that you can't address data that is outside the dataset you work with. Especially when you use more exotic expressions, the chances increase that you'll select data that comes from a source you didn't want to get data from. Separating such data with a namespace solves this problem.

Even if you have a document that consists of only elements in the same vocabulary, declaring the namespace is still a good idea. The best way to do so is to declare the namespace as the default namespace so that you don't have to type the namespace prefix for each element. When you process the document with a stylesheet that might process documents with different namespaces, the prefix in the stylesheet can differ from the prefix in the original document, as long as the namespace name is identical.

You also can use namespaces deliberately to be able to mix data. This way, you can mix data that has a grouping of some sort, so you can address the separate groups as a single group instead of having to spell out each data item. Such mixing of namespaces is shown in Listing 21.6.

LISTING 21.6 XML Document Mixing Namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<shop:basket xmlns:shop="http://www.example.com/xmlns/shop"
             xmlns:product="http://www.example.com/xmlns/products">
  <product:product product:ID="234" product:description="Bordeaux"
                product:price="50.00" shop:quantity="1"/>
  <product:product product:ID="123" product:description="Brie"
                product:price="99.95" shop:quantity="3"/>
</shop:basket>
```

ANALYSIS Listing 21.6 mixes namespaces to keep apart information from a shop and the inventory. If you want to select only the product information of the first product, you can use the following expression:

```
/shop:basket/product:product[1]/@product
```

This expression selects only the attributes in the `product` namespace. The attributes from the `shop` namespace are ignored. Now consider Listing 21.7.

LISTING 21.7 XML Document from Listing 21.6 Without Namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<basket>
  <product ID="234" description="Bordeaux" price="50.00" quantity="1"/>
  <product ID="123" description="Brie" price="99.95" quantity="3"/>
</basket>
```

Analysis Listing 21.7 shows the same information as Listing 21.6, but without the namespaces. Now no distinction exists between the data that belongs to the product itself or to the shop. If you want to get only the product information of the first product, you have to use either of the following expressions:

```
/basket/product[1]/@ID | /basket/product[1]/@description |
/basket/product[1]/@price
```

or

```
/basket/product[1]/@*[name() != 'quantity']
```

The latter expression is more versatile because it enables you to add product data, which will still be matched by that expression. The former expression, which is quite lengthy, selects only the specific attributes `ID`, `description`, and `price`.

Design Tools

XML design is similar to database design: Both require application-specific analysis that no tool provides. The reason for this is simple: XML is a flexible data format in which one format isn't necessarily worse than another. Design tools for applications and databases all work around the premise that an application is structured in a certain way. This is, of course, true, and applications that use XML as a data source also conform to the same structure. The underlying XML structure, however, is the domain of data modeling tools. All current data modeling tools are geared toward database design rather than XML design. XML design therefore is more or less still considered as an art. The preceding sections gave you insight into the *tools* that you, as the artist, have to work with. With time, experience will teach you how to best use these tools.

XML Schemas also can help you in designing XML structures. XML Schemas define vocabularies and XML structures. The primary concern of a Schema is validation of a document; however, a Schema is self-documenting. By using a stylesheet, you can gather all kinds of information from a Schema. You can download a stylesheet that documents a Schema from `http://msdn.microsoft.com/downloads/sample.asp?url=/msdn-files/027/000/539/msdncompositedoc.xml`.

Michael Corning is a pioneer in the field of application design and programming based on XML Schemas. The method of programming he promotes is called Schema-Based Programming (SBP), which he has closely linked to a design framework called the Model-View-Controller framework (MVC). This framework separates the data, the view, and the interaction control into three separate pieces. Corning argues that the whole application builds on the data, or the Model, if you will. Different views of that data, which are controlled by the Controller, enable interaction with the user. This idea is similar to three-tiered architectures in distributed applications, where the data is separated from the logic, which in turn is separated from the display. With MVC, this separation is different, and concentrates around the XML data model. You can find out more details about SBP and MVC at `http://www.aspalliance.com/mcorning/`. This site also contains downloadable source code for different implementations of an SBP/MVC application.

Designing XSLT

In the preceding section, you learned about the issues involved in designing XML documents. Some of the discussion was related to how easy or hard it is to perform tasks in XSLT. Most of those ideas are equally applicable to solutions using the Document Object Model (DOM) to access XML data. The next step is to look at XSLT itself and examine the considerations for designing a stylesheet. Although this topic is very much linked to the design of XML documents, these considerations apply to most stylesheets, regardless of the XML structure.

XSLT Design Considerations

When you design stylesheets, one of the most important goals you want to accomplish is that you can alter sections of a stylesheet without affecting other sections. Related to that goal is the possibility of reusing sections of stylesheets you create in other stylesheets. When you design stylesheets as part of larger systems, reusing sections becomes even more important because you don't have to reinvent the wheel each time. It also guarantees that across your application or applications the same data is processed consistently; especially in a Web site where the formatting needs to be consistent for all the pages, this is important. Mechanisms that can help you are variables, attribute-sets, templates, and so on, but when do you use which? The following section answers that question for you. As with XML design, the answers are not universal truths, but just guidelines.

Setting Up a Stylesheet Base

When you start a stylesheet, you start with the foundation. This foundation is the XML prolog, which tells any parser or processor that it is dealing with XML and what the encoding method of the document is. Next is the `xsl:stylesheet` element, which is also straightforward, unless you use extensions, in which case you have to declare the namespaces involved. Although you aren't required to declare namespaces until you actually use them, including any namespace declarations in the `xsl:stylesheet` element is good practice. This way, you can make sure that all developers can see at once which namespaces a stylesheet processes. Any elements in the source documents that use a namespace not declared in the stylesheet will not be processed.

Caution

The `version` attribute should always have the number of the current World Wide Web Consortium (W3C) Recommendation or lower. Using version numbers that haven't achieved Recommendation status yet is not a good idea. Only when a version has become a Recommendation are all elements and behaviors final.

Other important parts of your stylesheet base are the elements that deal with output encoding, whitespace handling, and so on. For text output, adding these elements is simple because you can specify only the encoding. Specifying the media type doesn't make sense in most situations. When you're creating text output, the best thing you can do is strip all nonsignificant whitespace from the source document and insert linefeeds and so on yourself. This way, you have 100% control over what the output will look like. To achieve this, your stylesheet should start with the following code:

```
<xsl:output method="text" encoding="UTF-8" />
<xsl:strip-space elements="*" />
```

The preceding code uses UTF-8 encoding, which is probably the most common. Unless you do something really out of the ordinary, or are working on a system that doesn't support it, keep it that way.

When creating XML or HTML output, you have many more options. Fortunately, these options aren't all very interesting in most cases. What is important is that you always specify the version you want to create. For XML, specifying the version doesn't make much sense now, but creating applications isn't just about here and now; XML is going to be around in the future, and that means versions might change.

The `cdata-section-elements` attribute of the `xsl:output` element is important when you have designed a document to contain CDATA sections. When you create a document that has to conform to the design of that XML structure, you need to specify the CDATA section. Although you can specify it later in the game, adding it before you do anything else is good practice.

When you're setting up a stylesheet base, a smart move is to add a template that matches all elements but does nothing. This template makes sure that data from unmatched elements isn't sent to the output, so when you use `xsl:apply-templates`, only elements that are explicitly matched produce output. The base for XML output therefore will look something like Listing 21.8.

LISTING 21.8 Stylesheet Base for XML Output

```
 <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" version="1.0" encoding="UTF-8" />

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

  <xsl:template match="*" />
</xsl:stylesheet>
```

Matching, Calling, or Iteration?

Whether you should use matching, calling, or iteration is the most important question when you're designing stylesheets. Your decisions will have a lasting effect on your application.

Matching is the key concept around which XSLT was designed, and it has some major advantages over iteration. The fact that XSLT was designed for matching doesn't necessarily mean that matching is faster than iteration, but you can safely assume that matching will be at least as fast as iteration. More importantly, templates are units of coding that can easily be changed. You could argue that the code inside an `xsl:for-each` element is equally changeable, but that is not quite true because the element is embedded in a template. More important is the fact that an iteration is bound to the template it is used in, so it is sensitive to the context of the template it is used in. A template, on the other hand, can be used independently of context, which means that it can be reused in separate sections of your stylesheet or application. If you want to enforce context on the template, you can do so by using the `select` attribute of the `xsl:apply-templates` element and different modes, if necessary.

Does this mean you should never use iteration? No, most certainly not. Every advantage has a disadvantage. In the case of matching templates, the advantage of being generic is also its disadvantage. If you want to do something specific only to the current context, using `xsl:for-each` is a good choice. Using different types of matching in that case would only complicate matters. Matching the contents of a variable, for instance, when you're working with multiple documents is tricky. When you iterate through a variable's content, however, using data that is outside the scope of the variable is much easier. With templates, you would have to use parameters to get around this problem.

If the question is "matching or iteration?" where does calling templates fit in? Called templates have two functions: One is to break functionality of a large template into smaller pieces. If you're using matching, this probably doesn't happen much because the templates are already pretty lean. With iteration and calculation, the situation is different, however. The other function of a called template is to solve problems that you can't solve with matching or iteration. These problems require a template to act as a function of some sort. With variables and parameters, you can make a template return data based on the value of one or more parameters. Although this is also possible when matching templates, you have much less control over the actual result. Recursive solutions, for instance, require called templates with parameters. Don't grab for recursion too quickly. You should treat it as a last resort if all else fails. In many cases, you also can find a solution using matching or iteration, but it might be less obvious. Recursion puts a strain on the processor that you want to avoid, especially if the source data is large. In that case, hundreds of recursive calls can occur, and some processors might not be able to handle them.

Variables and Attribute-sets

Variables are a fact of life in some applications, particularly those that process multiple documents. Variables are well suited for dynamic data, data that depends on the source document, the current context and scope, and so on. In all these cases, no alternatives exist. When you use matching on the contents of a variable, it is important that you keep in mind that the context changes from the source document to the variable. This means that the data from the source document can't be accessed, not even through a key. I have stated this point before, but it is so important that it doesn't hurt to tell you again. I have lost a lot of time in projects because I forgot that I was matching a variable, so my expressions selecting data from the source document drew a blank.

When it comes to static global data, you have an alternative for variables: attribute-sets. Attribute-sets are much more flexible than variables when it comes to adding values as attributes to an element because you can create attribute-sets from other attribute-sets and override the data in some attributes. The downside to attribute-sets is that the names of the attributes have to be known beforehand. When you use a variable to store a data value, you can create differently named attributes from that same value. Whether doing so is a good idea is debatable, however, because it goes against keeping stylesheets and XML documents consistent.

Another drawback of attribute-sets is that they can contain only attributes, so if you want to insert entire elements or element structures, you need to use variables anyway.

Multifile Stylesheets

I don't have much to say about multifile stylesheets other than what I already stated on Day 13, "Working with Multifile Stylesheets," except that using them is a good idea if your stylesheets become larger and different stylesheets have templates in common. Think about creating stylesheet libraries that contain templates you use throughout your application. Such a library can save you a great deal of time. By using `xsl:apply-imports`, you can also create incremental templates. A template that you import is then used from a template matching the same node. The template from the importing stylesheet inherits the functionality of the imported template and can add functionality to it.

Error Handling

Error handling in XSLT is hardly needed. If your stylesheet is syntactically correct, it will work with any source data. The only exception occurs when you use extension functions because they don't have the same constraints as XSLT functions. Error handling in XSLT is therefore more a concern in the sense that a stylesheet doesn't have the correct result if the source data is different than expected. For instance, number calculations yield the value `NaN` if a data value is missing. In many cases, you would actually like such a case to be treated as if the result were zero. This means that you have to check the value before the calculation takes place. The best way to do so is pre-emptively. Before you start to process a piece of data, check whether it conforms to the structure and values that you need. If the calculation needs to take place even if this isn't the case, create a variable that has the correct structure from scratch or from the data to be processed. That way, you're sure that the result is always correct. Listing 18.8 used this mechanism. The relevant section of Listing 18.8 is shown in Listing 21.9

LISTING 21.9 Partial Stylesheet Showing Pre-emptive Error Handling

```
<xsl:variable name="points">
  <xsl:choose>
    <xsl:when test="$result/points[@team = $id]">
      <for><xsl:value-of
                select="$result/points[@team = $id]/@for" /></for>
      <against><xsl:value-of
                    select="$result/points[@team = $id]/@against" />
      </against>
      <xsl:value-of select="$result/points[@team = $id]" />
    </xsl:when>
    <xsl:otherwise>
      <for>0</for><against>0</against>0
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

ANALYSIS Line 3 in Listing 21.8 checks whether the needed `points` element exists. If it does, the values are taken from it; otherwise, the values created on line 12 are all zero. The whole result is stored in a variable that can be used later in calculation. Because the values are all finite number values, you don't have to worry about a calculation yielding `NaN` because the `points` element checked on line 3 doesn't exist.

XSLT Design Do's and Don'ts

The previous sections discussed the differences between various options you have when designing stylesheets. When to apply which option is something you need to learn from experience. You will find that the more you work with XSLT, the more options you will have used once or twice. You will most certainly find out what works well, at least for you. I indeed have found that some solutions work better for me than others, and with that, I have developed a few Do's and Don'ts for myself that might help you.

- Use templates and matching as much as possible. Matching is at the core of XSLT and therefore works best.
- Make your templates as small as possible, without breaking them up into called templates. Smaller templates are easier to maintain, will force you to keep templates simple, and are easier to reuse.
- Use attribute-sets and imported stylesheets to increase the ability to reuse functionality.
- Don't use multiple source documents unless you absolutely have to. Using multiple documents complicates things, so if you don't need them, don't use them.
- Use only local variables and parameters, if possible. Local variables don't suffer much from scoping problems; global variables do and are therefore not handy in situations in which you reuse functionality.
- Don't use recursion unless there is no other way to solve a problem. Recursion puts a strain on the processor, and a processor is much more likely to fail than with matching and iteration.
- Always use `xsl:output`, `xsl:strip-space`, and `xsl:preserve-space` to control the output format and whitespace. If you don't explicitly set the conditions for output, not all processors will produce the same output.
- Never assume that your expression will always yield a value if it has to. XML documents in the real world are likely to be flawed, so you should check that the values you expect are really there. Only if you're sure that a document is validated by a Schema or DTD can you leave out such checks.

- Always create (and use) a stylesheet base. This will immediately restrict the number of mistakes you can make.
- Don't assume that your application will always stay the same. Design an application so that it can be extended, either by you or by others.

Summary

In this last lesson, you learned that designing XML and XSLT is mostly about experience. Although there are general rules, satisfying all conditions that result in well-designed XML and XSLT documents is impossible because some of the conditions are in conflict.

The most important aspects of XML and XSLT design are defining the hierarchy of the XML document and using matching in a stylesheet. These aspects go hand in hand because a good hierarchy makes it easier to match nodes. When you need to split up an XML source into several files, you will have to compromise on this hierarchy and use reference values. Doing so will have an adverse effect on the simplicity of your stylesheets. Even so, if you can still solve your problems with matching instead of resorting to iteration, you should try to do so because matching templates ensure that you have a stylesheet that consists of independent units.

Q&A

Q I come from a database background, and I'm used to working with database design tools. Can I use these tools for designing XML?

A Yes. Databases aren't suited for hierarchical data, however, so you need to be aware of that limitation when you design your documents. If you plan to separate documents similarly to database tables, the tools might be a big help.

Q I'm used to working with an object-oriented design tool. Can I use it to design XML?

A The answer is debatable. XML structures can resemble object-oriented structures, but most often they do not. Using such a tool is like trying to draw a picture while wearing a straightjacket.

Q Is it better to design XML first and then XSLT, or are they linked so much that I need to do them together?

A The answer depends on the situation. However, it never hurts to keep in mind what impact a certain XML design will have on stylesheets you might create for it.

Workshop

This workshop tests whether you understand all the concepts you learned today. It is helpful to know and understand the answers before starting tomorrow's lesson. You can find the answers to the quiz questions and exercises in Appendix A.

Quiz

1. True or False: Matching is better than iteration.
2. True or False: With a hierarchical relationship, you don't need reference values to refer to data in other elements.
3. Why would you use attributes instead of elements?
4. What is the biggest problem when you work with multiple source documents?
5. Why would you use a namespace in a document that uses a single vocabulary?

Exercise

1. Create an efficient XML document for the following data:

```
Products
Red wine: Bordeaux
red wine: Ruby Cabernet
White wine: Soave
Red wine: Chianti
Red wine: Merlot
Cheese: Camembert
Cheese: Gouda
Cheese: Brie
Cheese: Mozarella
Cheese: Feta

Order 1
Client: John Doe
Items: 6 Ruby Cabernet wines
       4 Chianti wines
       2 Brie cheeses

Order 2
Client: Michiel van Otegem
Items: 4 Bordeaux wines
       12 Merlot wines
       2 Camembert cheeses
       5 Mozzarella cheeses
```

WEEK 3

In Review

In this third and final week, you have learned more about XSLT using XSLT in real-world applications. You also have learned about design issues involved in creating real-world applications. You learned that although XSLT isn't a high-end programming language when it comes to computations, you can still do computations on source data, such as aggregating and performing statistical analysis.

You now have a solid grasp of all that XSLT has to offer. As you have seen, XSLT is very powerful in certain areas and can match any other language in that way. XSLT enables you to perform complex transformations with little effort. The same transformations would be very hard in most other languages. This is, of course, not surprising because XSLT is specifically designed to operate on XML, whereas languages such as C++, Java, and Visual Basic are more generic in their use of data sources.

Overview of Bonus Project 3

This week's Bonus Project aims at combining several of the points you have learned over the last seven days. Of course, not everything covered will be used here because the Bonus Project is about a situation that may occur in a real-world application. It is unlikely that all that has been discussed will find its way into one small project. You will, however, see namespaces and computation.

Note

This Bonus Project requires Internet Explorer 6.0 or higher, or Internet Explorer 5.0 or higher with version 3.0 or higher of the MSXML component installed in Replace mode. The easiest way to make sure that you can run the Bonus Project is by updating Internet Explorer through Windows Update, or `http://www.microsoft.com/windows/ie/default.asp`.

Creating a Shopping Basket in XSLT

This week's Bonus Project is a shopping basket that runs entirely in an XML- and XSLT-enabled browser. Because of that requirement, the shopping basket was written for Internet Explorer with MSXML 3.0 or higher. The code that runs the processor is written in JavaScript. When other browsers support XSLT processors it is easy to port to work on those browsers.

The end product of this Bonus Project is one large HTML document that consists of five major pieces:

- The HTML and JavaScript code that runs the shopping basket
- An XML island with the product XML
- An XML island with the shopping basket
- An XML island with the stylesheet responsible for displaying the shopping basket and the product file
- An XML island with the stylesheet performing calculations

New Term An *XML island* is an HTML element in Internet Explorer that can hold an XML document. In the HTML file, it looks as follows:

```
<xml id="myisland"><!--XML code here--></xml>
```

These XML islands hold all the data and XSLT. The rest of the code runs the processor, and so on.

The Product Data

The fact that this project is about a shopping basket means, of course, that there must be product data. The data chosen is closely related to the data from Bonus Project 2, with wines and cheeses. The product data is shown in Listing BP3.1.

Listing BP3.1 Product Data

```
1:  <prd:products xmlns:prd="http://www.aspnl.com/xmlns/products">
2:      <prd:product prd:ID="1" prd:description="Bordeaux"
3:           prd:price="9.95" prd:type="wine" prd:color="red" />
4:      <prd:product prd:ID="2" prd:description="Ruby Cabernet"
5:           prd:price="10.95" prd:type="wine" prd:color="red" />
6:      <prd:product prd:ID="3" prd:description="Soave"
7:           prd:price="8.95" prd:type="wine" prd:color="white" />
8:      <prd:product prd:ID="4" prd:description="Chianti"
9:           prd:price="11.95" prd:type="wine" prd:color="red" />
10:     <prd:product prd:ID="5" prd:description="Merlot"
11:          prd:price="7.95" prd:type="wine" prd:color="red" />
12:     <prd:product prd:ID="6" prd:description="Camembert"
13:          prd:price="3.95" prd:type="cheese" />
14:     <prd:product prd:ID="7" prd:description="Gouda"
15:          prd:price="4.95" prd:type="cheese" />
16:     <prd:product prd:ID="8" prd:description="Brie"
17:          prd:price="3.95" prd:type="cheese" />
18:     <prd:product prd:ID="9" prd:description="Mozzarella"
19:          prd:price="2.95" prd:type="cheese" />
20:     <prd:product prd:ID="10" prd:description="Feta"
21:          prd:price="1.95" prd:type="cheese" />
22: </prd:products>
```

Analysis Listing BP3.1 uses a namespace for the product data. The namespace prefix is `prd`, which is used for both elements and attributes. Each product has at least a unique identifier, description, and price. As you can see, it may have additional attributes. These additional attributes don't have to be the same, and each product doesn't need to have the same number of attributes. The wine on line 2 for instance has other attributes than the cheese on line 12. To keep the project simple, the additional data is ignored here, but the display stylesheet can easily be modified to show this information as well.

The Shopping Basket

The shopping basket's data—in essence, the items that somebody has placed in the shopping basket—is stored in XML. The shopping basket is the heart of the application because this data changes while the user moves through the shop. All the other data and stylesheets are static. The shop starts off with an empty shopping basket, as shown in Listing BP3.2.

Listing BP3.2 Empty Shopping Basket

```
<shop:basket xmlns:shop="http://www.aspnl.com/xmlns/shop"
             xmlns:prd="http://www.aspnl.com/xmlns/products">
</shop:basket>
```

ANALYSIS The empty shopping basket consists of only the `shop:basket` element. Note that two namespaces are declared, the first for the shopping basket data and the other for the product information. The latter is the same as the namespace used in the product data in Listing BP3.1. Suppose you put two bottles of Merlot and a Camembert cheese in the shopping basket; it would then look like Listing BP3.3.

LISTING BP3.3 Basket with Several Items

```
1: <shop:basket xmlns:shop="http://www.aspnl.com/xmlns/shop"
2:              xmlns:prd="http://www.aspnl.com/xmlns/products">
3:   <prd:product prd:ID="5" prd:description="Merlot"
4:     prd:price="7.95" prd:type="wine" prd:color="red" shop:quantity="2" />
5:   <prd:product prd:ID="6" prd:description="Camembert"
6:     prd:price="3.95" prd:type="cheese" shop:quantity="1" />
7: </shop:basket>
```

ANALYSIS Listing BP3.3 shows the shopping basket after several items have been placed in it. The product on line 10 of Listing BP3.1 has been copied over entirely, and the same goes for the product on line 12. Note that both elements have the additional attribute `shop:quantity` on lines 4 and 6. This attribute is part of the shopping basket data and therefore has a different namespace so that it can be distinguished from the product data itself. It is very important that *all* the product data is copied over. When the user is finished shopping, the whole result can be used in checking out, creating an invoice, shipping information, and so on. Using only product identifiers, this wouldn't be possible. In effect, the shopping basket is independent of any other data sources.

Displaying the Data

With the two data files, let's look at how this data will be displayed. Listing BP3.4 shows the stylesheet responsible for displaying the data.

LISTING BP3.4 Stylesheet Responsible for Displaying the Product Data and the Shopping Basket

```
<xsl:stylesheet version=" 1.0"exclude-result-prefixes="shop prd"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         xmlns:shop="http://www.aspnl.com/xmlns/shop"
         xmlns:prd="http://www.aspnl.com/xmlns/products">

   <xsl:output method="html" omit-xml-declaration="yes" />

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

LISTING BP3.4 Continued

```

  <xsl:template match="prd:products">
    <h1>Products</h1>
    <table border="1">
      <tr>
        <th>ID</th>
        <th>description</th>
        <th>price</th>
        <th></th>
      </tr>
      <xsl:for-each select="//prd:product">
        <tr>
          <td><xsl:value-of select="@prd:ID" /></td>
          <td><xsl:value-of select="@prd:description" /></td>
          <td>US $ <xsl:value-of select="@prd:price" /></td>
          <td><a><xsl:attribute name="href">#</xsl:attribute>
                 <xsl:attribute name="onclick">
                   <xsl:text>return addProduct(</xsl:text>
                   <xsl:value-of select="@prd:ID" />
                   <xsl:text>, 1)</xsl:text>
                 </xsl:attribute>
                 <xsl:text>add to cart</xsl:text></a>
          </td>
        </tr>
      </xsl:for-each>
    </table>
    <a href="#" onclick="return showBasket()">show basket</a>
  </xsl:template>

  <xsl:template match="shop:basket">
    <form id="basketform">
      <h1>Basket</h1>
      <table border="1">
        <tr>
          <th>Quantity</th>
          <th>ID</th>
          <th>Item</th>
          <th>Price per item</th>
          <th>Tax per item</th>
          <th>Product total +tax</th>
          <th>Delete</th>
        </tr>
        <xsl:for-each select="//prd:product">
          <xsl:call-template name="basketproduct" />
        </xsl:for-each>
      </table>
      <br />
      <input type="submit" value="Update basket"
             onclick="return updateBasket()" />
```

LISTING BP3.4 Continued

```
60:      </form>
61:      <a href="#" onclick="return showProducts()">show products</a>
62:    </xsl:template>
63:
64:    <xsl:template name="basketproduct">
65:      <xsl:variable name="taxrate" select="0.06" />
66:      <xsl:variable name="tax" select="@prd:price * $taxrate" />
67:      <xsl:variable name="producttotal"
68:                    select="@shop:quantity * (@prd:price + $tax)" />
69:      <tr>
70:        <td>
71:          <input type="text" size="2" name="update{@prd:ID}"
72:                 value="{@shop:quantity}" />
73:        </td>
74:        <td><xsl:value-of select="@prd:ID" /></td>
75:        <td><xsl:value-of select="@prd:description" /></td>
76:        <td><xsl:value-of select="@prd:price" /></td>
77:        <td><xsl:value-of
78:                 select="format-number($tax, '#,##0.00')" /></td>
79:        <td><xsl:value-of
80:                 select="format-number($producttotal, '#,##0.00')" />
81:        </td>
82:        <td>
83:          <input type="checkbox">
84:            <xsl:attribute name="name">
85:              <xsl:text>delete</xsl:text>
86:              <xsl:value-of select="@prd:ID" />
87:            </xsl:attribute>
88:          </input>
89:        </td>
90:      </tr>
91:    </xsl:template>
92:  </xsl:stylesheet>
```

ANALYSIS Listing BP3.4 has a double function: It is used to display both the product data and the contents of the shopping basket. Because the whole thing is already embedded in an HTML file, you don't have to worry about the outer HTML elements such as `HTML` and `BODY`. The template on line 12 is responsible for displaying the product data. It matches the `prd:products` element. Basically, it creates a table with each row showing a product. Each table cell shows product information. Line 21 iterates through the products and creates the table rows. Lines 26–32 are very important for the functionality of the output. These lines are responsible for creating a link that calls a JavaScript function that adds the product to the shopping basket. Line 28 tells the browser to call the `addProduct` JavaScript function, with line 29 inserting the identifier of the product that needs to be added to the shopping basket. Line 30 passes the quantity of the product to be added; for this project, the quantity is always 1. Listing BP3.5 shows what the HTML looks like when it is created for Listing BP3.1.

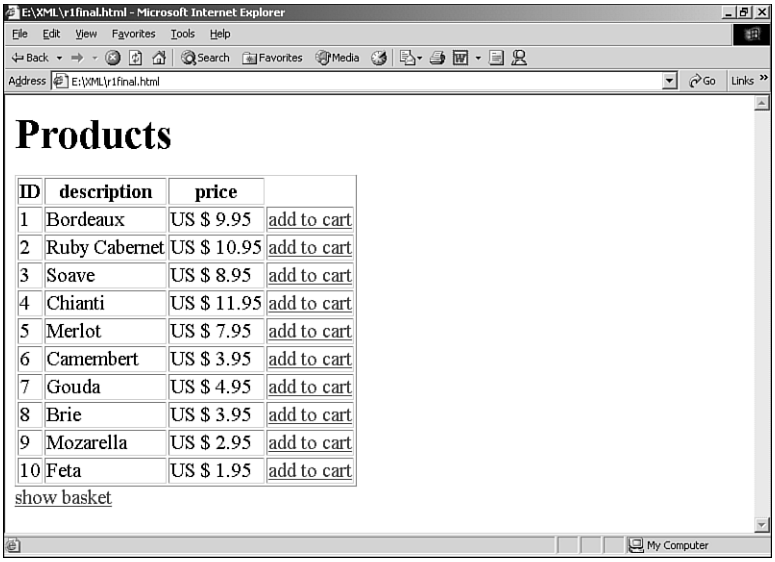
OUTPUT **LISTING BP3.5** Partial Listing of HTML Displaying the Product Data

```
1:  <H1>Products</H1>
2:  <TABLE border=1>
3:  <TBODY>
4:  <TR>
5:  <TH>ID</TH>
6:  <TH>description</TH>
7:  <TH>price</TH>
8:  <TH></TH></TR>
9:  <TR>
10: <TD>1</TD>
11: <TD>Bordeaux</TD>
12: <TD>US $ 9.95</TD>
13: <TD><A onclick="return addProduct(1, 1)"#">add to cart</A></TD></TR>
14: <TR>
15: <TD>2</TD>
16: <TD>Ruby Cabernet</TD>
17: <TD>US $ 10.95</TD>
18: <TD><A onclick="return addProduct(2, 1)" href="#">add to cart</A>
➥</TD></TR>
19: <TR>
20: <TD>3</TD>
21: <TD>Soave</TD>
22: <TD>US $ 8.95</TD>
23: <TD><A onclick="return addProduct(3, 1)" href="#">add to cart</A>
➥</TD></TR>
24: <!--additional products left out-->
25: </TBODY></TABLE><A onclick="return showBasket()" href="#">show basket</A>
```

ANALYSIS Listing BP3.5 shows only a partial result. For clarity, many of the products have been left out. Lines 4–8 display the table header row; then each product covers five lines. The last line, such as line 13, is of most interest. This is an a element, which is a hyperlink. It contains an attribute named `onclick`. The value of that attribute calls a JavaScript function in the HTML document that must add the product for which the link was clicked. Figure BP3.1 shows what this product list looks like in a browser.

In Listing BP3.4, the template on line 40 is responsible for displaying the products in the shopping basket. This template doesn't differ much from the template that displays the products in the product data. Line 54 invokes a called template for each product, to make the template a little shorter and handier to work with. The called template starts on line 59.

Line 71 inserts a text box with the number of items you want to have. If you want to change the number of items, you can change the number in that text box and update the basket by using the button inserted on line 59. Just like the links for the products, this button invokes a script function in the HTML. Line 83 creates a check box that you can check if you want to remove a product altogether. The resulting HTML is shown in Listing BP3.6


E:\XML\r1final.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\XML\r1final.html
Go Links
My Computer

Products

ID	description	price	
1	Bordeaux	US $ 9.95	add to cart
2	Ruby Cabernet	US $ 10.95	add to cart
3	Soave	US $ 8.95	add to cart
4	Chianti	US $ 11.95	add to cart
5	Merlot	US $ 7.95	add to cart
6	Camembert	US $ 3.95	add to cart
7	Gouda	US $ 4.95	add to cart
8	Brie	US $ 3.95	add to cart
9	Mozarella	US $ 2.95	add to cart
10	Feta	US $ 1.95	add to cart

show basket

FIGURE BP3.1
Listing BP3.5 when viewed in a browser.

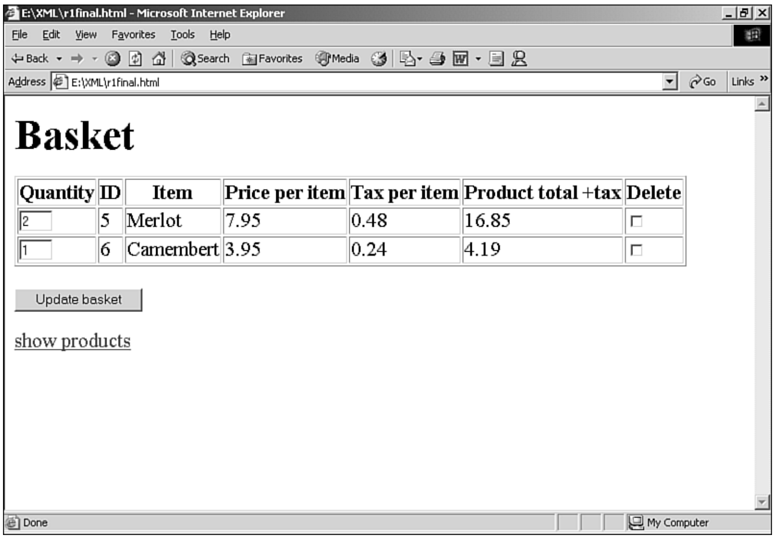
OUTPUT **LISTING BP3.6** HTML Displaying the Shopping Basket

```
<FORM id=basketform>
<H1>Basket</H1>
<TABLE border=1>
<TBODY>
<TR>
<TH>Quantity</TH>
<TH>ID</TH>
<TH>Item</TH>
<TH>Price per item</TH>
<TH>Tax per item</TH>
<TH>Product total +tax</TH>
<TH>Delete</TH></TR>
<TR>
<TD><INPUT size=2 value=2 name=update5></TD>
<TD>5</TD>
<TD>Merlot</TD>
<TD>7.95</TD>
<TD>0.48</TD>
```

LISTING BP3.6 Continued

```
<TD>16.85</TD>
<TD><INPUT type=checkbox name=delete5></TD></TR>
<TR>
<TD><INPUT size=2 value=1 name=update6></TD>
<TD>6</TD>
<TD>Camembert</TD>
<TD>3.95</TD>
<TD>0.24</TD>
<TD>4.19</TD>
<TD><INPUT type=checkbox name=delete6></TD></TR></TBODY></TABLE><BR>
<INPUT onclick="return updateBasket()" type=submit value="Update basket">
</FORM>
<A onclick="return showProducts()" href=" #">show products</A>
```

ANALYSIS The HTML in Listing BP3.6 shows a table embedded in an HTML form. The form consists of a button to update the basket, text boxes to change the quantity of each product, and a check box to delete a product. If you have multiple items in the basket, you can change the quantities you want and possibly check items to be deleted. These changes are committed and the basket recalculated when you click the button to update the basket. You can see what this form looks like in Figure BP3.2.


E:\XML\r1final.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\XML\r1final.html
Go Links
Done
My Computer

Basket

Quantity	ID	Item	Price per item	Tax per item	Product total +tax	Delete
2	5	Merlot	7.95	0.48	16.85	☐
1	6	Camembert	3.95	0.24	4.19	☐

Update basket

show products

FIGURE BP3.2

Listing BP3.6 when viewed in a browser.

Updating the Basket

NEW TERM When updating the basket, you have no way of knowing how many parameters you will need. To get around this problem, you use an *updategram* to actually update the shopping basket. An updategram is an XML tree fragment that is passed as such to the stylesheet, so you can pass any number of values. A sample updategram is shown in Listing BP3.7.

LISTING BP3.7 Updategram to Update the Shopping Basket

```
<updates>
  <delete ID="1" />
  <delete ID="4" />
  <update ID="3" quantity="10" />
  <update ID="7" quantity="2" />
</updates>
```

ANALYSIS The `delete` elements in Listing BP3.7 indicate which elements need to be deleted. Those that need to be updated are indicated by the `update` elements. For each product, both the ID and quantity are passed.

If you add only one product, an `add` element will suffice, as follows:

```
<add ID="5" quantity="1" />
```

The preceding code is still a child element of the `updates` element, so theoretically you can add, delete, and update with one updategram. This capability is handy if you choose to create a different implementation where you register the changes and, for instance, post an updategram back to the server. Based on the updategram and the current shopping basket, a new basket is created with the stylesheet in Listing BP3.8.

LISTING BP3.8 Stylesheet Creating New Shopping Basket

```
1:    <xsl:stylesheet version="1.0"
2:      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3:      xmlns:shop="http://www.aspnl.com/xmlns/shop"
4:      xmlns:prd="http://www.aspnl.com/xmlns/products">
6:
7:      <xsl:output method="xml" omit-xml-declaration="yes" />
8:
9:      <xsl:param name="products" select="empty" />
10:     <xsl:param name="updates" select="empty" />
11:
12:      <xsl:template match="/">
13:       <shop:basket>
14:         <xsl:call-template name="addproducts" />
15:         <xsl:call-template name="updateproducts" />
```

Listing BP3.8 Continued

```
    </shop:basket>
  </xsl:template>

   <xsl:template name="addproducts">
     <xsl:for-each select="$updates//add">
       <xsl:variable name="ID" select="@ID" />
      <xsl:variable name="quantity" select="@quantity" />
      <prd:product>
        <xsl:choose>
          <xsl:when test="/shop:basket/prd:product/@prd:ID = $ID">
            <xsl:copy-of select="/shop:basket/prd:product[./@prd:ID
                                 = $ID]/@prd:*" />
            <xsl:attribute name="shop:quantity">
            <xsl:value-of select="/shop:basket/prd:product[./@prd:ID
                                 = $ID]/@shop:quantity + $quantity" />
            </xsl:attribute>
          </xsl:when>
          <xsl:otherwise>
            <xsl:copy-of select="$products//prd:product[./@prd:ID
                                 = $ID]/@*" />
            <xsl:attribute name="shop:quantity">
          <xsl:value-of select="$quantity" />
            </xsl:attribute>
          </xsl:otherwise>
        </xsl:choose>
      </prd:product>
    </xsl:for-each>
  </xsl:template>

  <xsl:template name="updateproducts">
    <xsl:for-each select="/shop:basket/prd:product">
      <xsl:choose>
        <xsl:when test="@prd:ID = $updates//add/@ID" />
        <xsl:when test="@prd:ID = $updates//delete/@ID" />
        <xsl:when test="@prd:ID = $updates//update/@ID">
          <prd:product>
            <xsl:copy-of select="@prd:*" />
            <xsl:attribute name="shop:quantity">
              <xsl:value-of select="$updates//update[@ID
                                    = current()/@prd:ID]/@quantity" />
            </xsl:attribute>
          </prd:product>
        </xsl:when>
        <xsl:otherwise>
          <xsl:copy-of select="." />
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Analysis Listing BP3.8 processes the shopping basket document. In addition, the product data and updategram are passed to the parameters on lines 9 and 10. The product data could have been accessed with the `document()` function as well, but the choice in this project was to create one large file that the browser could download. In other scenarios, using the `document()` function is probably a better option.

Note MSXML allows you to pass an XML DOM document as a parameter. You can't pass the text of a tree fragment as a parameter because it will be output escaped first. Most processors don't allow passing DOM documents, so both the updategram and product data need to be loaded using the `document()` function when using those processors.

The template on line 12 creates the root element for the new shopping basket and then calls the templates responsible for adding products to the shopping basket or updating the current products in the basket. The template on line 19 iterates through all the `add` elements in the updategram and adds the corresponding product to the basket. Line 25 checks whether the product is already in the basket, in which case the product is altered instead of added. If the product exists, line 26 copies all the product data using the expression

```
/shop:basket/prd:product[./@prd:ID = $ID]/@prd:*"
```

This expression selects the product data from the shopping basket for the product that corresponds to the correct ID value. All the attributes are copied, only from the namespace of the product. Because the quantity has to be changed, the `shop:quantity` attribute is not copied, but handled by lines 28–31 instead. If the product doesn't exist in the basket yet, line 34 copies the product data from the `products` parameter, and lines 36–38 create the `shop:quantity` attribute.

The template on line 45 deals with the `update` and `delete` elements in the updategram. It iterates through all the elements in the shopping basket and checks whether the current element has a corresponding `delete` or `update` element. Line 48 checks whether there is a corresponding `add` element, in which case nothing happens because the element is already processed. Line 49 checks whether there is a `delete` element, in which case nothing happens because the element shouldn't be inserted into the new shopping basket. Line 50 checks whether there is an `update` element, in which case line 52 copies all the product data from the old shopping basket, with line 53 updating the quantity. If no `add`, `update`, or `delete` element appears in the updategram, line 60 just copies the entire item in the shopping basket to the new shopping basket.

Invoking the Processor

This project so far is all about XML and XSLT. You don't want the user messing around with a command-line processor, though, so you need to automate invoking the processor as well. This means you need to invoke the processor programmatically, as shown in Listing BP3.9.

LISTING BP3.9 Script Code Invoking the Processor

```
<script language="javascript">
var xslUpdate = null;

function body_onload() {
  var xslDoc = new ActiveXObject("MSXML2.FreeThreadedDOMDocument.3.0");
  xslUpdate = new ActiveXObject("MSXML2.XSLTemplate.3.0");
  xslDoc.async = false;
  xslDoc.load(update.XMLDocument);
  xslUpdate.stylesheet = xslDoc;

   showProducts();
}

function doUpdate(strUpdates) {
  var xmlUpdates = new ActiveXObject("MSXML2.DOMDocument.3.0");
  xmlUpdates.loadXML(strUpdates);
  var xmlInput = new ActiveXObject("MSXML2.DOMDocument.3.0");
  xmlInput.load(basket);

  var xslProc = xslUpdate.createProcessor();
  xslProc.addParameter("products", products);
  xslProc.addParameter("updates", xmlUpdates);
  xslProc.input = xmlInput;
  xslProc.output = basket;
  xslProc.transform();
}

function addProduct(ID, qty) {
  var strUpdates = "<updates><add ID='" + ID + "' quantity='"
  strUpdates = strUpdates + qty + "' /></updates>"
  doUpdate(strUpdates);
  showBasket();
  return false;
}

function updateBasket() {
  var strUpdates = "<updates>";
  for(var i = 0; i < document.forms[0].length - 1; i++) {
    if(document.forms[0].elements[i].type = 'text') {
      var ID = parseInt(document.forms[0].elements[i].name.substring(6));
```

LISTING BP3.9 Continued

```
41:       if(document.forms[0].elements[i+1].checked) {
42:         strUpdates = strUpdates + "<delete ID='" + ID + "'/>";
43:       } else {
44:         strUpdates = strUpdates + "<update ID='" + ID + "' ";
45:         strUpdates = strUpdates + "quantity='"
46:         strUpdates = strUpdates +
➥parseInt(document.forms[0].elements[i].value);
47:         strUpdates = strUpdates + "'/>"
48:       }
49:       i++;
50:     }
51:   }
52:   strUpdates = strUpdates + "</updates>";
53:   doUpdate(strUpdates);
54:   showBasket();
55:   return false;
56: }
57:
58: function showBasket() {
59:   result.innerHTML = basket.transformNode(display.XMLDocument);
60:   return false;
61: }
62:
63: function showProducts() {
64:   result.innerHTML = products.transformNode(display.XMLDocument);
65:   return false;
66: }
67: </script>
```

ANALYSIS Listing BP3.9, which is all JavaScript, uses several objects that are provided by the MSXML parser/processor component. It is beyond the scope of this book to really dive into the code because doing so requires good knowledge of JavaScript, but in broad terms you need to know what's going on. The functions `showBasket` and `showProducts`, on lines 58 and 63, speak for themselves. To show the resulting HTML, they set the HTML inside an HTML `div` element named `result`. On line 59, `basket` refers to the XML island that contains the basket XML. The processor is invoked for this XML island using the `transformNode` method. The argument of this function is the stylesheet that is responsible for display. It is passed from its own XML island named `display`. Line 64 does exactly the same for the products. The `addProduct` and `updateBasket` functions on lines 28 and 36 create an updategram in the `strUpdates` string. The updategram string is passed to the `doUpdate` function on line 14. This function creates an XML DOM document from that string, which is passed as a parameter to the stylesheet on line 22. Line 21 passes the product data to the stylesheet. Line 24 makes sure that the output is sent to the basket object. Because this object can't be processed and serve as a result container at the same time, line 18 creates a temporary object that is a copy of the current basket and becomes the document to be processed. Line 25 finally invokes the processor to perform the transformation.

Note

Listing BP3.10 shows the entire project file that you can run from the browser. You can download this file from the publisher's Web site.

LISTING BP3.10 Complete HTML Document

```
<script language="javascript">
var xslUpdate = null;

function body_onload() {
  var xslDoc = new ActiveXObject("MSXML2.FreeThreadedDOMDocument");
  xslUpdate = new ActiveXObject("MSXML2.XSLTemplate");
  xslDoc.async = false;
  xslDoc.load(update.XMLDocument);
  xslUpdate.stylesheet = xslDoc;

  showProducts();
}

function doUpdate(strUpdates) {
  var xmlUpdates = new ActiveXObject("MSXML2.DOMDocument.3.0");
  xmlUpdates.loadXML(strUpdates);
  var xmlInput = new ActiveXObject("MSXML2.DOMDocument.3.0");
  xmlInput.load(basket);

  var xslProc = xslUpdate.createProcessor();
  xslProc.addParameter("products", products);
  xslProc.addParameter("updates", xmlUpdates);
  xslProc.input = xmlInput;
  xslProc.output = basket;
  xslProc.transform();
}

function addProduct(ID, qty) {
  var strUpdates = "<updates><add ID='" + ID + "' quantity='"
  strUpdates = strUpdates + qty + "' /></updates>"
  doUpdate(strUpdates);
  showBasket();
  return false;
}
```

LISTING BP3.10 Continued

```
function updateBasket() {
  var strUpdates = "<updates>";
  for(var i = 0; i < document.forms[0].length - 1; i++) {
    if(document.forms[0].elements[i].type = 'text') {
      var ID = parseInt(document.forms[0].elements[i].name.substring(6));
      if(document.forms[0].elements[i+1].checked) {
        strUpdates = strUpdates + "<delete ID='" + ID + "'/>";
      } else {
        strUpdates = strUpdates + "<update ID='" + ID + "' ";
        strUpdates = strUpdates + "quantity='"
        strUpdates = strUpdates + parseInt
➥(document.forms[0].elements[i].value);
        strUpdates = strUpdates + "'/>"
      }
      i++;
    }
  }
  strUpdates = strUpdates + "</updates>";
  doUpdate(strUpdates);
  showBasket();
  return false;
}

function showBasket() {
  result.innerHTML = basket.transformNode(display.XMLDocument);
  return false;
}

function showProducts() {
  result.innerHTML = products.transformNode(display.XMLDocument);
  return false;
}
</script>
<html>
<body onload="body_onload()">

<div id="result"></div>

<xml id="products">
  <prd:products xmlns:prd="http://www.aspnl.com/xmlns/products">
    <prd:product prd:ID="1" prd:description="Bordeaux"
         prd:price="9.95" prd:type="wine" prd:color="red" />
    <prd:product prd:ID="2" prd:description="Ruby Cabernet"
         prd:price="10.95" prd:type="wine" prd:color="red" />
    <prd:product prd:ID="3" prd:description="Soave"
         prd:price="8.95" prd:type="wine" prd:color="white" />
    <prd:product prd:ID="4" prd:description="Chianti"
         prd:price="11.95" prd:type="wine" prd:color="red" />
    <prd:product prd:ID="5" prd:description="Merlot"
```

LISTING BP3.10 Continued

```
          prd:price="7.95" prd:type="wine" prd:color="red" />
     <prd:product prd:ID="6" prd:description="Camembert"
          prd:price="3.95" prd:type="cheese" />
     <prd:product prd:ID="7" prd:description="Gouda"
          prd:price="4.95" prd:type="cheese" />
     <prd:product prd:ID="8" prd:description="Brie"
          prd:price="3.95" prd:type="cheese" />
     <prd:product prd:ID="9" prd:description="Mozzarella"
          prd:price="2.95" prd:type="cheese" />
     <prd:product prd:ID="10" prd:description="Feta"
          prd:price="1.95" prd:type="cheese" />
   </prd:products>
 </xml>

 <xml id="basket">
   <shop:basket xmlns:shop="http://www.aspnl.com/xmlns/shop"
                xmlns:prd="http://www.aspnl.com/xmlns/products">
   </shop:basket>
 </xml>

 <xml id="display">
   <xsl:stylesheet version="1.0"exclude-result-prefixes="shop prd"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:shop="http://www.aspnl.com/xmlns/shop"
          xmlns:prd="http://www.aspnl.com/xmlns/products">

     <xsl:output method="html" omit-xml-declaration="yes" />

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

     <xsl:template match="prd:products">
       <h1>Products</h1>
       <table border="1">
         <tr>
           <th>ID</th>
           <th>description</th>
           <th>price</th>
           <th></th>
         </tr>
         <xsl:for-each select="//prd:product">
           <tr>
             <td><xsl:value-of select="@prd:ID" /></td>
             <td><xsl:value-of select="@prd:description" /></td>
             <td>US $ <xsl:value-of select="@prd:price" /></td>
             <td><a><xsl:attribute name="href">#</xsl:attribute>
                    <xsl:attribute name="onclick">
                      <xsl:text>return addProduct(</xsl:text>
```

LISTING BP3.10 Continued

```
                  <xsl:value-of select="@prd:ID" />
                  <xsl:text>, 1)</xsl:text>
                </xsl:attribute>
                <xsl:text>add to cart</xsl:text></a>
         </td>
       </tr>
     </xsl:for-each>
   </table>
   <a href="#" onclick="return showBasket()">show basket</a>
 </xsl:template>

 <xsl:template match="shop:basket">
   <form id="basketform">
     <h1>Basket</h1>
     <table border="1">
       <tr>
         <th>Quantity</th>
         <th>ID</th>
         <th>Item</th>
         <th>Price per item</th>
         <th>Tax per item</th>
         <th>Product total +tax</th>
         <th>Delete</th>
       </tr>
       <xsl:for-each select="//prd:product">
         <xsl:call-template name="basketproduct" />
       </xsl:for-each>
     </table>
     <br />
     <input type="submit" value="Update basket"
            onclick="return updateBasket()" />
   </form>
   <a href="#" onclick="return showProducts()">show products</a>
 </xsl:template>

 <xsl:template name="basketproduct">
   <xsl:variable name="taxrate" select="0.06" />
   <xsl:variable name="tax" select="@prd:price * $taxrate" />
   <xsl:variable name="producttotal"
                 select="@shop:quantity * (@prd:price + $tax)" />
   <tr>
     <td>
       <input type="text" size="2" name="update{@prd:ID}"
              value="{@shop:quantity}" />
     </td>
     <td><xsl:value-of select="@prd:ID" /></td>
     <td><xsl:value-of select="@prd:description" /></td>
     <td><xsl:value-of select="@prd:price" /></td>
     <td><xsl:value-of
```

LISTING BP3.10 Continued

```
                  select="format-number($tax, '#,##0.00')" /></td>
          <td><xsl:value-of
                  select="format-number($producttotal, '#,##0.00')" />
          </td>
          <td>
            <input type="checkbox">
              <xsl:attribute name="name">
                <xsl:text>delete</xsl:text>
                <xsl:value-of select="@prd:ID" />
              </xsl:attribute>
            </input>
          </td>
        </tr>
      </xsl:template>
    </xsl:stylesheet>
  </xml>

  <xml id="update">
    <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:shop="http://www.aspnl.com/xmlns/shop"
      xmlns:prd="http://www.aspnl.com/xmlns/products">

      <xsl:output method="xml" omit-xml-declaration="yes" />

      <xsl:param name="products" select="empty" />
      <xsl:param name="updates" select="empty" />

      <xsl:template match="/">
        <shop:basket>
          <xsl:call-template name="addproducts" />
          <xsl:call-template name="updateproducts" />
        </shop:basket>
      </xsl:template>

      <xsl:template name="addproducts">
        <xsl:for-each select="$updates//add">
          <xsl:variable name="ID" select="@ID" />
          <xsl:variable name="quantity" select="@quantity" />
          <prd:product>
            <xsl:choose>
              <xsl:when test="/shop:basket/prd:product/@prd:ID = $ID">
                <xsl:copy-of select="/shop:basket/prd:product[./@prd:ID
                                     = $ID]/@prd:*" />
                <xsl:attribute name="shop:quantity">
                  <xsl:value-of select="/shop:basket/prd:product[./@prd:ID
                                     = $ID]/@shop:quantity + $quantity" />
                </xsl:attribute>
              </xsl:when>
```

LISTING BP3.10 Continued

```
            <xsl:otherwise>
              <xsl:copy-of select="$products//prd:product[./@prd:ID
                                     = $ID]/@*" />
              <xsl:attribute name="shop:quantity">
                <xsl:value-of select="$quantity" />
              </xsl:attribute>
            </xsl:otherwise>
          </xsl:choose>
        </prd:product>
      </xsl:for-each>
    </xsl:template>

    <xsl:template name="updateproducts">
      <xsl:for-each select="/shop:basket/prd:product">
        <xsl:choose>
          <xsl:when test="@prd:ID = $updates//add/@ID" />
          <xsl:when test="@prd:ID = $updates//delete/@ID" />
          <xsl:when test="@prd:ID = $updates//update/@ID">
            <prd:product>
              <xsl:copy-of select="@prd:*" />
              <xsl:attribute name="shop:quantity">
                <xsl:value-of select="$updates//update[@ID
                                        = current()/@prd:ID]/@quantity" />
              </xsl:attribute>
            </prd:product>
          </xsl:when>
          <xsl:otherwise>
            <xsl:copy-of select="." />
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>
</xml>

</body>
</html>
```

APPENDIX A

Answers to Quiz Questions and Exercises

Answers for Day 1

Answers to Quiz Questions

1. True. Although XSLT operates on XML documents, it can generate any output that has a character-based format.
2. False. As long as the URI points to the proper location for XSLT, the actual name of the namespace is irrelevant. Note that the namespace URI needs to be exactly right; otherwise, the processor does not recognize your stylesheet as XSLT.
3. You run XSLT by using a processor. The processor reads the XML and XSLT and then applies the XSLT to the XML, generating output.
4. In a data-driven programming model, certain code is executed when a certain piece of data is encountered. Hence, the execution path is determined by the sequence and type of data that the program works on.

5. With declarative languages you tell the computer what you want to happen. In other languages, you tell the computer how to do something.
6. You can use many different tools. A simple text editor will do, but more sophisticated tools such as special XML editors and XSLT debuggers make the job easier.

Solution to Exercise

1. If you execute the files you created on different parsers, the output should be the same. After all, they implement the same specification. Any differences in output are caused by the freedom the specification gives in certain areas. The output should look like this:

```
My cat is called Max.


My parrot is called Peter.
Peter is red.
```

Answers for Day 2

Answers to Quiz Questions

1. False. The prolog is optional.
2. False. The simplified stylesheet syntax implies the existence of one template matching the document root. A template cannot contain other templates.
3. A template matches through a rule defined in the `match` attribute.
4. The `value-of` element is needed to get data from elements in the source XML.
5. Because an element contains text and child elements, outputting the value of that element yields the text of all ancestor elements.

Solution to Exercise

1. Your code should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="books">
My books:
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match="book">
    <xsl:value-of select="text()" />
  </xsl:template>

</xsl:stylesheet>
```

Answers for Day 3

Answers to Quiz Questions

1. True. `self` refers to the context node.
2. True. A predicate can contain a location path to test against. In today's lesson, this concept was covered only in theory.
3. This expression automatically selects the first element that matches it. Hence, the result is `Grilled Salmon`.
4. The output is an empty string. Because of the default precedence rules, the document doesn't have a 10th `dish` element. The processor first drills down and counts the `dish` nodes within the context of its parent.
5. The output is `To Top It Off`. The default precedence is overruled by the parentheses.

Solution to Exercise

1. Your code should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="menu/*">
    <xsl:value-of select="@title" />
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="dish">
    <xsl:value-of select="text()" /> $<xsl:value-of select="@price" />
  </xsl:template>

</xsl:stylesheet>
```

Answers for Day 4

Answers to Quiz Questions

1. True. Both attributes do not interfere. The former is used when `apply-templates` is used; the latter when `call-template` is used.
2. False. The more specific the match expression, the higher the priority. If the expression does not act on the current context, the priority may be different. This is also true if the expression uses additional axes.
3. No. Templates that have no mode are matched only when no mode is active.
4. The `select` attribute can hold an expression that selects certain nodes. The processor uses only these nodes to match against templates.
5. When nodes are selected, a node-set that can be acted on sequentially is created. When nodes are matched, templates are invoked in the sequence in which the nodes appear in the source document.

Solutions to Exercises

1. You can achieve this result in several ways. For instance, you can use named templates, modes, or `apply-templates` with a `select` attribute. The code for using modes is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="cars">
Models:
    <xsl:apply-templates mode="models" />
Manufacturers:
    <xsl:apply-templates mode="manufacturers" />
Years:
    <xsl:apply-templates mode="years" />
  </xsl:template>

  <xsl:template match="car" mode="models">
    -<xsl:value-of select="@model" />
  </xsl:template>

  <xsl:template match="car" mode="manufacturers">
    -<xsl:value-of select="@manufacturer" />
  </xsl:template>
  <xsl:template match="car" mode="years">
    -<xsl:value-of select="@year" />
  </xsl:template>
</xsl:stylesheet>
```

2. Like Exercise 1, this exercise has several solutions. In this case, a select expression is added to the `apply-templates` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
Models:
    <xsl:apply-templates select="cars/car/model" />
Manufacturers:
    <xsl:apply-templates select="cars/car/manufacturer" />
Years:
    <xsl:apply-templates select="cars/car/year" />
  </xsl:template>

  <xsl:template match="model">
    -<xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="manufacturer">
    -<xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="year">
    -<xsl:value-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

A stylesheet for both Listings 4.4 and 4.14 could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
Models:
    <xsl:apply-templates select="//model|//car/@model" />
Manufacturers:
    <xsl:apply-templates select="//manufacturer|//car/@manufacturer" />
Years:
    <xsl:apply-templates select="//year|//car/@year" />
  </xsl:template>

  <xsl:template match="model|@model">
    -<xsl:value-of select="." />
  </xsl:template>
  <xsl:template match="manufacturer|@manufacturer">
    -<xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="year|@year">
    -<xsl:value-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 5

Answers to Quiz Questions

1. False. As far as the processor is concerned, the characters are the same. There is a difference only for the underlying parser, but it presents the same character to the processor, regardless of how it appeared in the source document. You cannot determine whether a character was inserted as is or using output escaping.
2. False. The `xsl:attribute` tag can be used inside a template, with `xsl:copy`, or to insert attributes of a literal element.
3. The `name()` function returns the name of the context node.
4. With shallow copy, only the context element is copied; with deep copy, the tree fragment that starts with the context element is copied, including any attributes.
5. Yes. The former creates an element with the name of the context node. If it is an attribute, a new element is created with that attribute's name. With `xsl:copy`, if the context node is an attribute, an attribute is created, not an element.

Solution to Exercise

1. Your stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="cars">
    <xsl:copy>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>
  <xsl:template match="car">
    <xsl:copy>
      <xsl:apply-templates select="@*" />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="@*">
    <xsl:element name="{name()}">
      <xsl:value-of select="." />
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 6

Answers to Quiz Questions

1. No. `xsl:for-each` needs to be contained in a template. It does not have to be a direct child of the `xsl:template` element. It also may be nested in all the other elements covered so far.
2. No. `xsl:otherwise` is optional. You need to use `xsl:otherwise` only if you have a general case that you want to handle.
3. With selecting data, operations are performed in sequence on the data selected. With matching data, the next operation is determined by which element comes next in the document tree. Hence, selecting data is a pull data model, whereas matching data is a push data model.
4. With `xsl:if`, each block where the test expression evaluates to `true` is executed, regardless if any of the previous blocks were executed. With `xsl:when`, only one block is executed. The first block in which the test expression evaluates to `true` is executed; any others are not.
5. No. They are completely the same. `xsl:if` and predicate expressions are interchangeable.

Solutions to Exercises

1. Your stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <cars>
      <xsl:for-each select="/cars/car[@year='1999']">
```

```
        <xsl:copy-of select="." />
      </xsl:for-each>
      <xsl:for-each select="/cars/car[@year='2000']">
        <xsl:copy-of select="." />
      </xsl:for-each>
    </cars>
  </xsl:template>
</xsl:stylesheet>
```

2. Your stylesheet might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <cars>
      <xsl:for-each select="/cars/car">
        <xsl:copy>
          <xsl:copy-of select="@*" />
          <xsl:choose>
            <xsl:when
                 test="@manufacturer='Ford' or @manufacturer='Chevrolet'">
              <xsl:attribute name="country">USA</xsl:attribute>
            </xsl:when>
            <xsl:when test="@manufacturer='Honda' or
➥@manufacturer='Toyota'">
            <xsl:attribute name="country">Japan</xsl:attribute>
            </xsl:when>
            <xsl:when test="@manufacturer='Volkswagen'">
              <xsl:attribute name="country">Germany</xsl:attribute>
            </xsl:when>
          </xsl:choose>
        </xsl:copy>
      </xsl:for-each>
    </cars>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 7

Answers to Quiz Questions

1. False. `xsl:output` is a top-level element. It can therefore be a child only of the `xsl:stylesheet` element.
2. True. These elements can be child elements only of the `xsl:stylesheet` element.

3. The characters that are not supported by the output encoding are output escaped following the XML rules for output escaping. If a text document is being created, output escaping is not possible, and an error is reported.
4. `car` has priority 0, and `foo:*` has priority -0.25; therefore, `car` has a higher priority because `car` is more specific.
5. `xsl:strip-space` works only on whitespace-only nodes. It does not remove any redundant whitespace in an element value with mixed content. `normalize-space` removes redundant whitespace from text values and mixed content values.

Solution to Exercise

1. You can employ several options to create the output, but the following output shows a very structured and neat approach:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

  <xsl:template match="cars">
    <xsl:text>List of cars&#xA;</xsl:text>
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="car">
    <xsl:text>-</xsl:text>
    <xsl:value-of select="@manufacturer" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="@model" />
    <xsl:text> (</xsl:text>
    <xsl:value-of select="@year" />
    <xsl:text>)&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 8

Answers to Quiz Questions

1. False. As long as the scope of two variables does not overlap, they can have the same name. When they do overlap, the scope more specific to the current context is accessible; the other is not.

2. True. That's why they're called global variables. The only exception to this rule occurs when a variable is created specific to a certain scope with the same name. Within that variable's scope, the global variable is inaccessible.
3. You can use either of the following two methods:

   ```
   <xsl:variable name="car">Ford Focus</xsl:variable>
   ```

   ```
   <xsl:variable name="car" select="'Ford Focus'" />
   ```

4. Using the `select` attribute, you can select only a value that is exactly the same as in the source XML. Using the `xsl:variable` element body, you can also construct variable values using XSLT elements.
5. No. You can create a variable that contains the value of an attribute, but you need to create either a simple variable or a variable consisting of a node-set or tree-fragment. An attribute by itself cannot exist.

Solution to Exercise

1. Your code might look like the following, but you achieve the same result in many other ways.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:output method="html" version="4.0" />

     <xsl:variable name="bgcolor">
       <body>#cccccc</body>
       <table>#ffffff</table>
       <row1999>#cccccc</row1999>
       <row2000>#ffffff</row2000>
       <row2001>#ffcccc</row2001>
     </xsl:variable>

     <xsl:template match="/">
       <html>
       <body bgcolor="{$bgcolor/body}">
         <xsl:apply-templates select="cars/models" />
       </body>
       </html>
     </xsl:template>

     <xsl:template match="models">
       <table bgcolor="{$bgcolor/table}" width="75%">
         <xsl:for-each select="model">
           <tr>
   ```

```
          <xsl:attribute name="bgcolor">
            <xsl:choose>
              <xsl:when test="@year = '1999'">
                <xsl:value-of select="$bgcolor/row1999" />
              </xsl:when>
              <xsl:when test="@year = '2000'">
                <xsl:value-of select="$bgcolor/row2000" />
              </xsl:when>
              <xsl:otherwise>
                <xsl:value-of select="$bgcolor/row2001" />
              </xsl:otherwise>
            </xsl:choose>
          </xsl:attribute>
          <xsl:call-template name="car" />
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

  <xsl:template name="car">
    <xsl:variable name="mfc" select="@manufacturer" />
    <td><xsl:value-of select="@name" /></td>
    <td><xsl:value-of
             select="/cars/manufacturers/manufacturer[@id = $mfc]/@name" />
     </td>
    <td><xsl:value-of select="@year" /></td>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 9

Answers to Quiz Questions

1. False. Parameters and variables share their naming rules. A variable and a parameter can have the same name, as long as their scopes do not collide.
2. False. Although this element is used infrequently, it is most certainly possible.
3. The value will be ignored because there is no way to address it.
4. You give a parameter a value so that it will have a default value that is to be used when no value is passed on to it.
5. No. Parameters must be defined either as top-level elements or as the first elements in a template. You cannot create a parameter as a child element of other elements, such as `xsl:if` or `xsl:for-each`.

Solution to Exercise

1. Your stylesheet should look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" version="1.0" encoding="UTF-8" />

  <xsl:param name="id" />
  <xsl:param name="dish" />
  <xsl:param name="price" />
  <xsl:param name="type" />

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

  <xsl:template match="menu">
    <xsl:copy>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="appetizers|entrees|desserts">
    <xsl:copy>
      <xsl:apply-templates />
      <xsl:if test="name() = $type">
        <xsl:call-template name="insertdish" />
      </xsl:if>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="dish">
    <xsl:copy-of select="." />
  </xsl:template>

  <xsl:template name="insertdish">
    <dish id="{$id}" price="{$price}">
      <xsl:value-of select="$dish" />
    </dish>
  </xsl:template>
</xsl:stylesheet>
```

If you want to add Chicken Curry as an entree, you can use the following command line (note the quotation marks around `Chicken Curry`, to deal with the space):

```
saxon 09list05.xml 09ex01.xsl type=entrees id=99 dish="Chicken Curry"
➥ price="11.95"
```

Answers for Day 10

Answers to Quiz Questions

1. True. The value of the node-set is first converted to a string, concatenating all the text nodes. This string is then converted to a Boolean, following the conversion rules.
2. True. Either of the two values is always converted into the other before the comparison is made. After that, the comparison can yield only `true` or `false`.
3. Positive and negative zero are approximations for expressions that yield values so close to zero that they can't be expressed as finite numbers anymore. If the values could be expressed as finite numbers, they would be either negative or positive.
4. Although positive and negative zero are approximations, zero is zero. In essence, they are the same value. Only when these values are used in an expression do they behave differently.
5. With variable `x` containing a NaN value, the expressions `string($x) = 'NaN'` and `$x != $x` return `true`.

Solution to Exercise

1. The following listing shows several conversions and double conversions, and comparisons between "strange" numbers:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    Infinity = Infinity? <xsl:value-of select="1 div 0 = 2 div 0" />
    -Infinity = -Infinity? <xsl:value-of select="-1 div 0 = -2 div 0" />
    0 = -0? <xsl:value-of select="0 = -0" />
    NaN = NaN? <xsl:value-of select="number('xyz') = number('abc')" />

    Conversions:
    <xsl:for-each select="datatypes/*">
       <xsl:value-of select="." /> (<xsl:value-of select="name()" />)
       number(): <xsl:value-of select="number(.)" />
       boolean(): <xsl:value-of select="boolean(.)" />
       string(): <xsl:value-of select="string(.)" />
       <xsl:choose>
         <xsl:when test="name() = 'number'">
           number(string()): <xsl:value-of select="number(string(.))" />
           number(boolean()): <xsl:value-of select="number(boolean(.))" />
```

```
          </xsl:when>
          <xsl:when test="name() = 'boolean'">
            boolean(string()): <xsl:value-of select="boolean(string(.))" />
            boolean(number())): <xsl:value-of select="boolean(number(.))" />
          </xsl:when>
          <xsl:when test="name() = 'string'">
            string(number()): <xsl:value-of select="string(number(.))" />
            string(boolean()): <xsl:value-of select="string(boolean(.))" />
          </xsl:when>
        </xsl:choose>
        <xsl:text>&#xA;    </xsl:text>
      </xsl:for-each>
    </xsl:template>

</xsl:stylesheet>
```

Answers for Day 11

Answers to Quiz Questions

1. False. The `contains()` function returns only a Boolean value indicating whether the searched-for string was found in the string that was searched.
2. True. Any function can be used only in a test or select expression. It cannot be used in other attributes or in XSLT element values.
3. The first `x` encountered is used in the function, so the result is `defxgh`.
4. There is no corresponding character for the `x`, so the result is `ab|de|gh`.
5. You can separate the format for positive and negative numbers with a semicolon. You can use another character as the separator if you define it with `xsl:decimal-format`.

Solutions to Exercises

1. The result should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:variable name="monthnames">
    <month number="1">January</month>
    <month number="2">February</month>
    <month number="3">March</month>
    <month number="4">April</month>
    <month number="5">May</month>
    <month number="6">June</month>
    <month number="7">July</month>
    <month number="8">August</month>
    <month number="9">September</month>
    <month number="10">October</month>
    <month number="11">November</month>
    <month number="12">December</month>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:call-template name="formatdate" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:call-template name="formattime" />
  </xsl:template>

  <xsl:template name="formatdate">
    <xsl:param name="datetime" select="." />
    <xsl:variable name="year" select="substring-before($datetime,'-')" />
    <xsl:variable name="month" select="number(substring($datetime,6,2))" />
    <xsl:variable name="day" select="substring($datetime,9,2)" />
    <xsl:text>Today is </xsl:text>
    <xsl:value-of select="$monthnames/month[@number=$month]" />
    <xsl:text> </xsl:text><xsl:value-of select="$day" />
    <xsl:text>, </xsl:text><xsl:value-of select="$year" />
    <xsl:text>.</xsl:text>
  </xsl:template>

  <xsl:template name="formattime">
    <xsl:param name="datetime" select="." />
    <xsl:variable name="time" select="substring-after($datetime,'T')" />
    <xsl:variable name="hours" select="substring($time,1,5)" />
    <xsl:variable name="seconds" select="substring($time,7,2)" />
    <xsl:variable name="zone" select="substring($time,9,5)" />
    <xsl:text>The time is </xsl:text>
    <xsl:value-of select="$hours" />
    <xsl:text> hours and </xsl:text><xsl:value-of select="$seconds" />
    <xsl:text> seconds in timezone </xsl:text>
    <xsl:value-of select="$zone" />
    <xsl:text>.</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

2. The files you create for this exercise depend on how creative you are. The following listings give you an idea of some of the approaches you can use.

Sample XML

```
<?xml version="1.0" encoding="UTF-8"?>
<numbers>
  <number>1234567.8910</number>
  <number>-1234567.8910</number>
</numbers>
```

Sample stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <xsl:decimal-format name="inf" infinity="&#8734;" NaN="Invalid" />
  <xsl:decimal-format name="EURO"
                      decimal-separator=","
                      grouping-separator="."
                      digit="@" />

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

  <xsl:template match="numbers">
    <xsl:apply-templates />
    <xsl:text>&#xA;===special values===&#xA;</xsl:text>
    <xsl:value-of select="format-number(1 div 0,'#,##0.00','inf')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number(-1 div 0,'#,##0.00','inf')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number('xyz','#,##0.00','inf')" />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="number">
    <xsl:text>&#xA;===</xsl:text>
    <xsl:value-of select="." />
    <xsl:text>===&#xA;</xsl:text>

    <xsl:value-of select="format-number(.,'#,##0.00')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number(.,'###0.0##')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number(.,'#,###.00')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number(.,'#0')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="format-number(.,'@.@@0,00','EURO')" />
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of
         select="format-number(.,'# @.@@0,00;# -@.@@0,00','EURO')" />
    <xsl:text>&#xA;===&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 12

Answers to Quiz Questions

1. False. Although this statement is true when you're sorting one type of element, it is not true when you have different elements to be sorted.
2. True. How these elements are numbered through or separately depends on the value of the `count` attribute.
3. Numbers sort differently than text. So, if you're sorting on numbers, you need to specify that you are doing so; otherwise, 10,000 will come before 200.
4. The context element is sorted in ascending order.
5. If you use this approach, you end up with noncomposite numbers because basically it is the same as using `level="single"`.

Solutions to Exercises

1. Your stylesheet should look something like the following. As you can see, the numbering is jumbled.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

  <xsl:template match="cars">
    <xsl:apply-templates>
      <xsl:sort select="@manufacturer" />
      <xsl:sort select="@model|@name" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="car|model">
      <xsl:number level="single" count="car|model" />
      <xsl:text> </xsl:text>
      <xsl:value-of select="@manufacturer" />
      <xsl:text> </xsl:text>
      <xsl:value-of select="@model|@name" />
      <xsl:text> </xsl:text>
      <xsl:value-of select="concat('(',@year,')')" />
      <xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

2. Your stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

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

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

  <xsl:template match="appetizers|entrees|desserts">
    <xsl:number level="single" format="A"
                count="appetizers|entrees|desserts" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="concat(@title,'&#xA;')" />
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="dish">
    <xsl:text> </xsl:text>
    <xsl:number level="single" format="i" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="." />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 13

Answers to Quiz Questions

1. False. Global variables can be defined only once, unless there is difference in import precedence. When you're including a stylesheet, the import precedence is the same.
2. True. Imports are always done at the beginning of a stylesheet. The stylesheet imported first has the lowest import priority; the last, the highest. A template in the importing stylesheet has an even higher priority.
3. The imported and importing stylesheets are merged as much as possible. If they have conflicting values, the import precedence determines which value is used.

4. No. All `xsl:import` elements have to come before any other top-level elements.
5. No. A local variable has a narrower scope than a global variable, so it always wins. Because templates themselves are not changed, the variable stays intact.

Solution to Exercise

1. Your stylesheet might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="13list04.xsl" />

  <xsl:template match="appetizers|entrees|desserts">
    <h2><xsl:value-of select="@title" /></h2>
    <xsl:apply-imports />
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 14

Answers to Quiz Questions

1. False. You can use the `document()` function in any expression.
2. True. Data in multiple documents is treated as data from the same document, as long as you use the `document()` function or address the data through a variable.
3. Nothing happens. The data is basically duplicated. You can access both of the datasets.
4. No. You also can specify the Base URI as part of the URI that you pass to open a file.
5. The context of the matching template is switched to that variable (or the other document, if you will). So, the root context is the root of that variable.

Solutions to Exercises

1. Your stylesheet should look like the following and should be applied to Listing 14.6:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" version="4.0" encoding="UTF-8" />
  <xsl:strip-space elements="*" />
  <xsl:variable name="cars" select="document('14list01.xml')" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:for-each select="manufacturer">
      <h2><xsl:value-of select="@name" /></h2>
      <p><i>Country: <xsl:value-of select="@country" /></i></p>
      <xsl:for-each
           select="$cars/cars/model[@manufacturer = current()/@id]">
        <ul>
          <li>
            <xsl:value-of select="@name" />
            <xsl:text> (</xsl:text>
            <xsl:value-of select="@year" />
            <xsl:text>)</xsl:text>
          </li>
        </ul>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

2. Your stylesheet should look like the following. Try it on samples from Lesson 13. Note that the `xsl:text` elements are included just to provide proper indentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="xsl:stylesheet">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="xsl:include">
    <xsl:variable name="doc" select="document(@href)" />
    <xsl:for-each select="$doc/xsl:stylesheet/child::*">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:copy-of select="." />
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 15

Answers to Quiz Questions

1. True. You can mix namespaces as you like. Note that you have to address these attributes with their namespace.
2. False. As long as the namespace name (the URI) is the same, which prefix you use is not important.
3. Namespaces are related to DTDs and Schemas in that they all deal with XML vocabularies. Namespaces, however, are not meant as validation mechanisms.
4. Elements without a namespace and with a namespace are disjoint. They are separate entities and therefore handled separately.
5. You can add the `exclude-result-prefixes` attribute to the stylesheet element and create a whitespace-separated list of prefixes you do not want to be carried over.

Solution to Exercise

1. Your stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:car="http://www.example.com/xmlns/car"
  xmlns:man="http://www.example.com/xmlns/manufacturer"
  exclude-result-prefixes="car man">

  <xsl:output method="html" version="4.0" />

  <xsl:template match="/">
    <html>
    <body>
      <h1>Auto show</h1>
      <xsl:apply-templates select="/car:cars/man:manufacturers" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="man:manufacturers">
    <xsl:for-each select="man:manufacturer">
      <h2><xsl:value-of select="@man:name" /></h2>
      <p><i>Country: <xsl:value-of select="@man:country" /></i></p>
      <xsl:for-each select="/car:cars/car:models/car:model[@man:id
➥ = current()/@man:id]">
        <ul>
          <li>
            <xsl:value-of select="@car:name" />
            <xsl:text> (</xsl:text>
            <xsl:value-of select="@car:year" />
            <xsl:text>)</xsl:text>
          </li>
        </ul>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 16

Answers to Quiz Questions

1. False. A key is defined only to quickly retrieve nodes. Whether the key is unique for each node is not relevant. If it is not, the `key()` function returns a node-set.
2. True. The value generated by the `generate-id()` function is based on the specific node it is used on. For that node, the value is always the same. Note that this value isn't necessarily the same between different processors. Most processors will generate the same values on subsequent runs.
3. The value of the first node in the node-set is converted to the same type and then compared, so if the value of the first node is equal to the number or the string, the node-set is equal to the number or string.
4. This answer depends on the processor. In a processor that implements a key with an internal index, storing the result in a variable makes no sense because the key already provides quick access. If there is no internal index, then storing the result makes sense because the processor will re-evaluate the expression used to create the key each time.
5. These values are, by definition, unique. This enables you to easily create an index so that the values can be retrieved easily.

Solution to Exercise

1. The code should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="UTF-8" />
  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="manufacturers">
    <xsl:copy>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="manufacturer">
    <xsl:copy>
      <xsl:attribute name="id">
        <xsl:value-of select="generate-id()" />
      </xsl:attribute>
      <xsl:attribute name="name">
        <xsl:value-of select="@name" />
      </xsl:attribute>
      <xsl:attribute name="country">
        <xsl:value-of select="@country" />
      </xsl:attribute>
      <xsl:attribute name="code">
        <xsl:value-of select="@id" />
      </xsl:attribute>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 17

Answers to Quiz Questions

1. False. If there are no parameters, you cannot pass any information that needs to be processed. You also can't call the same template with a smaller dataset, so there is no way to determine when the recursion should end.
2. True. The escape hatch is what stops the recursion. If the recursion doesn't end, the template keeps calling itself.
3. Not really. You have no way of controlling whether the correct template gets invoked. When the recursion is done on a node-set, it may still make sense, but especially when you're working with a single value, there is no way to match it with a template.
4. Sure. Because a recursive template is basically like any other template, it can return any data type.
5. You shouldn't use recursion when you can solve the same problem by using matching or iteration.

Solutions to Exercises

1. The resulting code should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:apply-templates select="names/name" />
  </xsl:template>

  <xsl:template match="name">
    <xsl:value-of select="substring-before(normalize-space(),' ')" />
    <xsl:text> </xsl:text>
    <xsl:call-template name="initial">
      <xsl:with-param name="fullname"
           select="substring-after(normalize-space(),' ')" />
    </xsl:call-template>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template name="initial">
    <xsl:param name="fullname" />
    <xsl:choose>
      <xsl:when test="contains($fullname,' ')">
        <xsl:value-of select="substring($fullname,1,1)" />
        <xsl:text>. </xsl:text>
        <xsl:call-template name="initial">
          <xsl:with-param name="fullname"
               select="substring-after($fullname,' ')" />
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$fullname" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

2. The resulting code should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes" encoding="UTF-8" />

  <xsl:template match="/">
    <xsl:apply-templates select="names/name" />
  </xsl:template>
  <xsl:template match="names">
    <xsl:copy>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="name">
    <xsl:copy>
      <xsl:call-template name="breakdown">
        <xsl:with-param name="fullname" select="normalize-space()" />
      </xsl:call-template>
    </xsl:copy>
  </xsl:template>

  <xsl:template name="breakdown">
    <xsl:param name="fullname" />
    <xsl:choose>
      <xsl:when test="contains($fullname,' ')">
        <firstname>
          <xsl:value-of select="substring-before($fullname,' ')" />
        </firstname>
        <xsl:call-template name="breakdown">
          <xsl:with-param name="fullname"
               select="substring-after($fullname,' ')" />
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <lastname><xsl:value-of select="$fullname" /></lastname>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 18

Answers to Quiz Questions

1. True. The `sum()` function can aggregate only a node-set with number values. Any operations that you need to perform on those nodes before they can be aggregated need to be performed before you use the `sum()` function. This means you have to create a variable that contains the nodes you want to aggregate if you need to perform other operations as well.
2. True. When you use matching on a variable, the context changes from the source document to the variable. The data in the source document is not accessible through absolute addressing.
3. `count(descendant::*)`

4. No. All operators work on numbers, Boolean values, or node-sets,
5. With other approaches, you either need to use the DOM from a programming language or parse the data from the XML source with the programming language "by hand." With XSLT, you don't have to learn additional languages.

Solution to Exercise

1. You can probably find several solutions to this problem. One of the stylesheets is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8" />
  <xsl:variable name="teams" select="//team" />

  <xsl:template match="/">
    <xsl:variable name="points">
      <xsl:apply-templates select="/competition/matches" />
    </xsl:variable>
    <xsl:variable name="ranking">
      <xsl:for-each select="/competition/teams/team">
        <points id="{@id}"
            for="{sum($points/points[@id = current()/@id]/@for)}"
            against="{sum($points/points[@id = current()/@id]/@against)}">
        <xsl:value-of select="sum($points/points[@id = current()/@id])" />
        </points>
      </xsl:for-each>
    </xsl:variable>
    <xsl:apply-templates select="$ranking/points">
      <xsl:sort select="." order="descending" />
      <xsl:sort select="@for - @against" order="descending" />
      <xsl:sort select="@for" order="descending" />
      <xsl:with-param name="teams" select="//team" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="points">
    <xsl:value-of select="$teams[@id = current()/@id]" />
    <xsl:text> </xsl:text><xsl:value-of select="." />
    <xsl:text> </xsl:text><xsl:value-of select="@for" />
    <xsl:text> </xsl:text><xsl:value-of select="@against" />
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
  <xsl:template match="matches">
    <xsl:for-each select="match">
      <xsl:choose>
        <xsl:when test="@homescore &gt; @awayscore">
          <points id="{@hometeam}" for="{@homescore}"
                  against="{@awayscore}">2</points>
          <points id="{@awayteam}" for="{@awayscore}"
                  against="{@homescore}">0</points>
        </xsl:when>
        <xsl:when test="@homescore &lt; @awayscore">
          <points id="{@hometeam}" for="{@homescore}"
                  against="{@awayscore}">0</points>
          <points id="{@awayteam}" for="{@awayscore}"
                  against="{@homescore}">2</points>
        </xsl:when>
        <xsl:otherwise>
          <points id="{@hometeam}" for="{@homescore}"
                  against="{@awayscore}">1</points>
          <points id="{@awayteam}" for="{@awayscore}"
                  against="{@homescore}">1</points>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 19

Answers to Quiz Questions

1. False. Processor extensions are extensions to specific processors.
2. False. The `extension-element-prefixes` attribute is required only for extension elements. Adding them doesn't hurt, however, so doing so is a good idea.
3. You need to declare a namespace that incorporates the full Java class name. You can then call the functions in the class by using the namespace prefix and function name.
4. You can't use extensions when your stylesheets need to run on any processor and any platform.
5. When a new version of XSLT that supports embedded script becomes the standard, minor changes to the stylesheet will make it processor independent. When you use vendor-specific processor extensions, making the stylesheet processor independent might not be so easy.

Solutions to Exercises

1. The resulting stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://icl.com/saxon"
  extension-element-prefixes="saxon"
  exclude-result-prefixes="saxon">

  <xsl:output method="text" encoding="utf-8" />

  <xsl:template match="/">
    <xsl:text>The largest number is </xsl:text>
    <xsl:value-of select="saxon:max(//number)" />
  </xsl:template>
</xsl:stylesheet>
```

2. The resulting stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt"
    xmlns:user="http://www.aspnl.com/xmlns/extensions">

  <msxsl:script language="javascript" implements-prefix="user">
    function max(nodelist) {
      var maxval = parseInt(nodelist.nextNode().text);
      for(var i = 1;i &lt; nodelist.length;i++) {
        var intval = parseInt(nodelist.nextNode().text);
        if(maxval &lt; intval) {
          maxval = intval;
        }
      }
      return maxval;
    }
  </msxsl:script>

  <xsl:template match="/">
     <xsl:value-of select="user:max(//number)"/>
  </xsl:template>
</xsl:stylesheet>
```

Answers for Day 20

Answers to Quiz Questions

1. False. The `xsl:fallback` element is suitable only as an alternative for constructions using the `element-available()` function. It has no bearing on functions, only elements.

2. False. The `xsl:fallback` element also can be used to deal with support for processor extension elements.
3. When elements with mixed values (such as elements and text) are indented, the whitespace in the text might change, changing the semantics of the document.
4. The `system-property-function()` can be used to gather information about the processor. This information can be used to alter processing.
5. No, not with SQL Server by itself. If you want to transform the XML data before you return it to the client, you need to install XSL ISAPI, which performs server-side XSLT transformations.

Solution to Exercise

1. This exercise has more than one solution. The solution here creates the files in the directory `c:\xml`. If you want to change this, change the value of the `dir` variable in the stylesheet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://icl.com/saxon"
  extension-element-prefixes="saxon"
  exclude-result-prefixes="saxon">

  <xsl:variable name="dir">c:\xml</xsl:variable>

  <xsl:output method="html" encoding="UTF-8" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <html>
    <body>
    <h1>News</h1>
    <xsl:apply-templates select="//newsitem" />
    </body>
    </html>
  </xsl:template>

  <xsl:template match="newsitem">
    <saxon:output href="{$dir}\news{@id}.html">
      <xsl:call-template name="createnewsfile" />
      <xsl:fallback />
    </saxon:output>
    <xsl:choose>
      <xsl:when test="element-available('saxon:output')">
        <a href="{$dir}\news{@id}.html">
          <xsl:value-of select="@title" />
        </a>
        <br />
      </xsl:when>
```

```
        <xsl:otherwise>
          <hr />
          <xsl:call-template name="createnewsitem" />
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

    <xsl:template name="createnewsitem">
      <h2><xsl:value-of select="@title" /></h2>
      <p><i><xsl:value-of select="@date" /></i></p>
      <p>
        <xsl:value-of select="." />
      </p>
    </xsl:template>

    <xsl:template name="createnewsfile">
      <html>
      <body>
      <xsl:call-template name="createnewsitem" />
      </body>
      </html>
    </xsl:template>
  </xsl:stylesheet>
```

Answers for Day 21

Answers to Quiz Questions

1. False. Both have their purposes; each is preferred in different situations.
2. True. The hierarchy defines the relationship between elements. You can use the hierarchy to address data in other elements related to the current node.
3. Attributes take less space and enforce the fact that there is only one value.
4. When you match data from a secondary document, the data from the primary document is unavailable.
5. You would use a namespace in this case because it uniquely identifies what the data is about. A stylesheet that matches multiple documents with different namespaces can distinguish the elements, even if they are all defined as the default namespace in the source document.

Solution to Exercise

1. The product data needs to be separated from the orders. The orders and the order details have a hierarchical relationship. The XML should have the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
<products>
  <product id="1" type="wine" color="red">Bordeaux</product>
  <product id="2" type="wine" color="red">Ruby Cabernet</product>
  <product id="3" type="wine" color="white" type="dry">Soave</product>
  <product id="4" type="wine" color="red" type="dry">Chianti</product>
  <product id="5" type="wine" color="red">Merlot</product>
  <product id="6" type="cheese">Camembert</product>
  <product id="7" type="cheese">Gouda</product>
  <product id="8" type="cheese">Brie</product>
  <product id="9" type="cheese">Mozzarella</product>
  <product id="10" type="cheese">Feta</product>
</products>
<orders>
  <order id="1">
    <client name="John Doe" />
    <orderdetail>
      <product id="2" qty="6" />
      <product id="4" qty="4" />
      <product id="8" qty="2" />
    </orderdetail>
  </order>
  <order id="2">
    <client name="Michiel van Otegem" />
    <orderdetail>
      <product id="1" qty="4" />
      <product id="5" qty="12" />
      <product id="6" qty="2" />
      <product id="9" qty="5" />
    </orderdetail>
  </order>
</orders>
</xml>
```

This page intentionally left blank

APPENDIX B

A Quick Reference of XSLT Elements and Functions

Throughout this book, you have learned about XSLT elements and functions. The lessons provide you with insight and samples about their usages, but because the lessons can be long, information is somewhat scattered. This reference helps you to quickly find information about the elements and functions in XSLT.

This reference does not discuss the elements and functions in depth. It serves as a quick reference to help you when you need to find a specific element or function, or need information on the values that attributes or arguments can have and what their purpose is.

Usage samples in this reference use italicized words. These words indicate that you need to fill in a value as the word indicates. For instance, `Expresssion` tells you that you need to fill in an expression yielding a result. You may encounter the following italicized terms:

- *`Character`*, *`string`*, *`number`*, *`node`*, or *`node-set`*, which indicates the type of data expected.
- *`Expression`*, which indicates an expression yielding a value.
- *`Format`*, which indicates a formatting pattern for a function.
- *`Name Test`*, which indicates an element name to be tested for.
- *`QName`*, which indicates a Qualified Name, or a name that is valid in XML, with or without a namespace prefix.
- *`Prefix`*, which indicates a namespace prefix.
- *`Pattern`*, which indicates a pattern identifying nodes in the source document or in a variable.
- *`URI`*, which indicates a Uniform Resource Indicator, mostly pointing to a file.
- *`Value`*, which indicates a value of any data type.

Values that are separated by a pipe symbol (`|`) are alternatives. You need to choose one of the alternatives.

XSLT Element Reference

This section describes the elements in XSLT in alphabetical order.

`xsl:apply-imports`

The `xsl:apply-imports` element is used to invoke templates in imported templates. The template invoked is the one that matches the match rule and mode of the current match template. This element allows you to use the functionality of a template that you have overridden by writing a new template matching the same nodes. It applies only when other stylesheets have been imported with `xsl:import`.

Attributes

This element has no attributes.

Content

This element is always empty.

Usage

```
<xsl:apply-imports />
```

Position

This element can be used only in a template body of a matched template. It cannot be used in a called template and has no effect if used inside an `xsl:for-each` element.

`xsl:apply-templates`

The `xsl:apply-templates` element is used to invoke templates for the selected nodes or, if no nodes are specified, the child elements of the context element.

Attributes

`select` (optional)	Expression used to select the node(s). If this attribute is omitted, the child elements of the context node are processed.
`mode` (optional)	The name of the processing mode. Only the templates with a matching mode name are used to process the selected nodes.

Content

Zero or more `xsl:sort` elements to define the sort order of the matched nodes, and zero or more `xsl:with-param` elements to pass parameters to other templates.

Usage

```
<xsl:apply-templates select="Expression" mode="QName">
```

Position

This element can be used anywhere in a template body.

`xsl:attribute`

The `xsl:attribute` element is used to add an attribute to an element in the output tree. The attribute is added to the element that is inserted as a parent of the `xsl:attribute` element. It also can be used to define an attribute in an attribute-set.

Attributes

`name`	Name for the attribute to be inserted. This can be an expression evaluating to a name.
`namespace` (optional)	Namespace URI for the attribute to be inserted. This can be an expression evaluating to a namespace URI.

Content

The value to be given to the attribute. This value needs to be a string but can be created using XSLT elements.

Usage

```
<xsl:attribute name="{QName}" namespace="{URI}">
```

Position

This element can be used anywhere in the template body. It also can be used in an `xsl:attribute-set` element.

`xsl:attribute-set`

The `xsl:attribute-set` element is used to define a named set of attributes that can be added to elements. An attribute-set can be added to an element in the output tree by using the `uses-attribute-sets` attribute.

Attributes

`name`	Name for the attribute-set.
`use-attribute-sets` (optional)	The names of attribute-sets to be incorporated into the attribute-set being defined. This should be a whitespace-separated list.

Content

Zero or more `xsl:attribute` elements.

Usage

```
<xsl:attribute-set name="QName" uses-attribute-sets="list-of-QNames" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:call-template`

The `xsl:call-template` element is used to invoke a named template. This is similar to a subroutine or procedure call in other programming languages. You can pass parameters to the invoked template with `xsl:with-param` elements.

Attributes

`name`	Name of the template to be invoked.

Content

Zero or more `xsl:with-param` elements.

Usage

```
<xsl:call-template name="QName">
```

Position

This element can occur anywhere in a template body.

`xsl:choose`

The `xsl:choose` element is used to define a choice between a number of alternatives. The alternatives are defined using any number of `xsl:when` elements and one `xsl:otherwise` element (optional).

Attributes

The element has no attributes.

Content

Zero or more `xsl:when` elements. Zero or one `xsl:otherwise` element.

Usage

```
<xsl:choose>
```

Position

This element can occur anywhere in a template body.

`xsl:comment`

The `xsl:comment` element is used to write a comment to the output. If the output is XML or HTML, the comment will be valid in either language.

Attributes

The element has no attributes.

Content

The comment to be inserted into the output. This comment can be a mixture of text and XSLT elements yielding text output.

Usage

```
<xsl:comment>comment</xsl:comment>
```

Position

This element can occur anywhere in a template body.

`xsl:copy`

The `xsl:copy` element is used to copy the context node to the output tree. It does not copy any of the attributes or descendant elements of the current node (shallow copy). If the node is an element, the namespace is also copied (if present).

Attributes

`use-attribute-sets` (optional)	The names of attribute-sets to be inserted with the copied element. This should be a whitespace-separated list.

Content

The element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element. The content is used only if the context node is a root node or an element.

Usage

<xsl:copy uses-attribute-sets="*list-of-QNames*">

Position

This element can occur anywhere in a template body.

xsl:copy-of

The `xsl:copy-of` element is used to copy the selected node or node-set to the output tree, including all its attributes and descendant elements (deep copy).

Attributes

`select`	Expression used to select the node(s) to be copied.

Content

This element is always empty.

Usage

<xsl:copy-of select="*Expression*">

Position

This element can occur anywhere in a template body.

xsl:decimal-format

The `xsl:decimal-format` element is used to define the characters used when a number is converted from a number to a string for output using the `format-number()` function.

Attributes

`name` (optional)	Name of the decimal format. If this attribute is omitted, the decimal format is used as the default.
`decimal-separator` (optional)	Character to be used to separate integers and fractions in a number. The default character is a period.

`grouping-separator` (optional)	Character to be used to separate groups of digits (for thousands, millions, and so on). The default character is a comma.
`infinity` (optional)	String to be used to represent the value `Infinity`. By default, the string used is `Infinity`.
`minus-sign` (optional)	Character to be used as a minus sign (to denote negative numbers). The default character is `-`.
`NaN` (optional)	String to be used to represent the value `NaN` (not a number). By default, the string used is `NaN`.
`percent` (optional)	Character to be used as a percentage sign. The default character is `%`.
`per-mille` (optional)	Character to be used to represent the per mill (per thousand) sign. The default character is `%`.
`zero-digit` (optional)	Character to be used to indicate a place where a leading zero is required in a formatting pattern. The default character is `0`.
`digit` (optional)	Character to be used to indicate a place where a digit is required in a formatting pattern. The default character is `#`.
`pattern-separator` (optional)	Character to be used to separate patterns for positive and negative numbers in a formatting pattern. The default character is a semicolon.

Content

This element is always empty.

Usage

```
<xsl:decimal-format name="QName"
                    decimal-separator="character"
                    grouping-separator="character"
                    infinity="string"
                    minus-sign="character"
                    NaN="string"
                    percent="character"
                    per-mille="character"
                    zero-digit="character"
                    digit="character"
                    pattern-separator="character" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:element`

The `xsl:element` element is used to insert an element into the output tree.

Attributes

`name` (optional)	The name for the element to be inserted into the output. The name can be generated from an expression.
`namespace` (optional)	The namespace for the element to be inserted into the output. The namespace can be generated from an expression.
`use-attribute-sets` (optional)	The attribute-sets to be inserted with the element. This should be a whitespace-separated list of attribute-sets.

Content

The element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

```
<xsl:element name="{QName}" namespace="{URI}"
             uses-attribute-sets="list-of-QNames">
```

Position

The only element that can occur anywhere in a template body.

xsl:fallback

The `xsl:fallback` element is used in situations in which XSLT elements that might not be supported by the processor are used. In such a situation, the `xsl:fallback` element can be inserted as an immediate child of the element in question. The body of the `xsl:fallback` element is executed instead of the parent element if the parent element is not supported.

Attributes

The element has no attributes.

Content

The element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

```
<xsl:fallback>
```

Position

This element can occur anywhere in a template body.

xsl:for-each

The `xsl:for-each` element is used to iterate through a selected node-set. The body of the `xsl:for-each` element is processed for each node in the selected node-set.

Attributes

`select`	Expression used to select the node(s) to be iterated.

Content

The element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

`<xsl:for-each select="`*`Expression`*`">`

Position

This element can occur anywhere in a template body.

xsl:if

The `xsl:if` element is used to process code conditionally. The body of the `xsl:if` element is processed only if the given condition is true.

Attributes

`test`	The expression that is used as a condition to process the body or not. The result of the expression should be a Boolean value (otherwise, the result is converted).

Content

The element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

`<xsl:if test="`*`Expression`*`">`

Position

This element can occur anywhere in a template body.

xsl:import

The `xsl:import` element is used to import other stylesheets into the stylesheet using the `xsl:import` element. The `xsl:import` element may occur in the imported stylesheet as well, creating a nesting of imported stylesheets. Any templates that are imported get an import precedence. Which template is invoked when a node is matched depends on the import precedence. Templates in the importing stylesheet have a higher import precedence than those in an imported stylesheet.

Attributes

`href`	The URI pointing to the stylesheet to be imported. This can be a relative or an absolute URI.

Content

This element is always empty.

Usage

```
<xsl:import href="URI" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element. In addition, `xsl:import` elements must precede any other top-level elements.

xsl:include

The `xsl:include` element is used to include other stylesheets into the stylesheet using the `xsl:include` element. The `xsl:include` element may occur in the included stylesheet as well, creating a nesting of included stylesheets. The elements for the included stylesheet(s) are placed in the including stylesheet at the position of the `xsl:include` element. The resulting stylesheet acts as if there is one stylesheet, and any precedence rules are applied as if it is one stylesheet.

Attributes

`href`	The URI pointing to the stylesheet to be included. It can be a relative or an absolute URI.

Content

This element is always empty.

Usage

```
<xsl:include href="URI" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:key`

The `xsl:key` element is used to create a named key that can later be used with the `key()` function. The `xsl:key` element, in conjunction with the `key()` function, allows quick access to elements based on a key value.

Attributes

`name`	The name of the key.
`match`	Pattern used to determine the nodes to which the key is applicable.
`use`	Expression used to determine the key of each of the nodes selected with the `match` attribute.

Content

This element is always empty.

Usage

```
<xsl:key name="QName" match="Pattern" use="Expression" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:message`

The `xsl:message` element is used to insert a message into the output, and optionally the execution of the stylesheet is stopped. The `xsl:message` element is generally used to report errors.

Attributes

`terminate` (optional)	Indicates whether to terminate execution of the stylesheet. The value can be either `yes` or `no` (default).

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

```
<xsl:message terminate="yes|no">message</xsl:message>
```

Position

This element can occur anywhere in a template body.

xsl:namespace-alias

The `xsl:namespace-alias` element is used to change the namespace (prefix) of nodes sent to the output so that it is different from the namespace (prefix) used in the stylesheet.

Attributes

`stylesheet-prefix`	Namespace prefix used in the stylesheet. The default namespace can be indicated with the value `#default`.
`result-prefix`	Namespace prefix to be used in the output. The default namespace can be indicated with the value `#default`.

Content

This element is always empty.

Usage

```
<xsl:namespace-alias stylesheet-prefix="prefix" result-prefix="prefix" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

xsl:number

The `xsl:number` element is used to insert sequential numbering for the context node in a node-set. It also can be used to output a number, similar to using the `format-number()` function.

Attributes

`level` (optional)	Determines whether the numbering is constricted to nodes with the same parent (`single`), to any level of the document (`any`), or if composite numbering should be used (`multiple`).
`count` (optional)	Pattern determining which nodes are counted to determine the sequence number of the context node.
`from` (optional)	Pattern determining from which point in the source document tree composite numbering should start.

`value` (optional)	Expression yielding a number to be formatted. If this expression is used, there is not sequential numbering.
`format` (optional)	Format string determining the format of the sequential number.
`lang` (optional)	Determines which language's numbering conventions should be used. The value should be a value valid for the `xml:lang` attribute.
`letter-value` (optional)	Distinguishes two different numbering conventions within the same language. The valid values are `alphabetical` or `traditional`.
`grouping-separator` (optional)	Character to be used to separate groups of digits (for thousands, millions, and so on). The default character is no character.
`grouping-size` (optional)	Determines the number of digits after which a grouping-separator character should be inserted.

Content

This element is always empty.

Usage

```
<xsl:number level="single|any|multiple"
            count="Pattern"
            from="Pattern"
            value="Expression"
            format="formatting string"
            language="xml:lang code"
            letter-value="alphabetical|traditional"
            grouping-separator="character"
            grouping-size="number" />
```

Position

This element can occur anywhere in a template body.

xsl:otherwise

The `xsl:otherwise` element is used to define an alternative choice in an `xsl:choose` element used when none of the `xsl:when` elements apply.

Attributes

This element has no attributes.

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

`<xsl:otherwise>`

Position

This element can occur only as a child of an `xsl:choose` element and can occur only once.

`xsl:output`

The `xsl:output` element is used to specify the format of the output. The format can be XML, HTML, text, or other formats supported by the processor.

Attributes

`method` (optional)	Determines the output format. The value can be `xml`, `html`, `text`, or a value other than these output formats that is supported by the processor.
`version` (optional)	Sets the version of the output method. This attribute is not applicable to the `text` output method.
`encoding` (optional)	Character encoding used for the output. Valid values are Unicode, ISO/IEC 10646, ISO 8859, or JIS X-0208-1997 character encoding. Note that all processors need to support UTF-8 or UTF-16 (not both), and are not obligated to support any others.
`omit-xml-declaration` (optional)	Determines whether the XML declaration should be inserted with XML output. Valid values are `yes` and `no`. This attribute is not applicable for HTML or text output.
`standalone` (optional)	Indicates that a standalone declaration should be included in the XML declaration, using the given value. Valid values are `yes` and `no`. This attribute is not applicable to the `text` output method.
`doctype-public` (optional)	Indicates that the public identifier should be included in the `DOCTYPE` declaration with the given value. This value should be a reference to a DTD. This attribute is not applicable to the `text` output method.
`doctype-system` (optional)	Indicates that the `system` identifier should be included in the `DOCTYPE` declaration with the given value. This attribute is not applicable to the `text` output method.
`cdata-section-elements` (optional)	Indicates the names of those elements whose text content should be output as `CDATA` sections. This should be a whitespace-separated list of elements. This attribute is not applicable to the `text` output method.

`indent` (optional)	Determines whether the output should be indented. Note that the indentation may differ from processor to processor and that a processor is not obligated to implement this feature. This attribute is not applicable to the `text` output method.
`media-type` (optional)	Indicates the media-type or MIME type to be associated with the output. This is applicable only in systems where the processor provides output to a system using these media-types. Most processors do not support this attribute.

Content

This element is always empty.

Usage

```
<xsl:output method="xml|html|text"
            version="versionnumber"
            encoding="string"
            omit-xml-declaration="yes|no"
            standalone=" yes|no "
            doctype-public="string"
            doctype-system="string"
            cdata-section-elements="list-of-QNames"
            indent=" yes|no "
            media-type="string" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:param`

The `xsl:param` element is used to define a parameter in a stylesheet (global parameter) or a template (local parameter).

Attributes

`name`	The name of the parameter.
`select` (optional)	Expression used to select the value for the parameter. The expression may yield a string, number, Boolean, node-set, or tree fragment. This value is used only if no value is received for the parameter from outside the stylesheet or template.

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element. If the `select` attribute is present, the element should be empty.

Usage

```
<xsl:param name="QName" select="Expression">
```

Position

This element can occur as a top-level element or as the first element(s) in a template body.

xsl:preserve-space

The `xsl:preserve-space` element is used to control the way whitespace-only nodes in the source document are handled. It is used in conjunction with the `xsl:strip-space` element.

Attributes

`elements`	Indicates the nodes in which whitespace-only nodes should be preserved. This should be a whitespace-separated list of names, possibly using wildcard characters.

Content

This element is always empty.

Usage

```
<xsl:preserve-space elements="list-of-name-tests" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

xsl:processing-instruction

The `xsl:processing-instruction` element is used to insert a processing instruction into the output tree.

Attributes

`name`	The name for the processing instruction.

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

```
<xsl:processing-instruction name="QName">
    processing instruction
</xsl:processing-instruction>
```

Position

This element can occur anywhere in a template body.

xsl:sort

The `xsl:sort` element is used to specify the sort order of a node-set. This element is used with `xsl:apply-templates` or `xsl:for-each`. Each `xsl:sort` element defines a sort key and order. You can use multiple `xsl:sort` elements in order of precedence.

Attributes

`select` (optional)	Expression defining the sort key. Normally, this is a string defining an element or attribute of the context node. If this attribute is omitted, the sort key is equal to `string(.)`.
`order` (optional)	Determines the order in which nodes are sorted. The value can be `ascending` (default) or `descending`.
`case-order` (optional)	Determines whether uppercase letters come before or after their lowercase counterparts. The value can be `upper-first` or `lower-first`. The default value is language dependent.
`lang` (optional)	Determines which language's sorting conventions should be used. The value should be a value valid for the `xml:lang` attribute.
`data-type` (optional)	Determines if the values are sorted as `text` (default), `number`, or based on a user-defined data type.

Content

This element is always empty.

Usage

```
<xsl:sort select="Expression"
          order="ascending|descending"
          case-order="upper-first|lower-first"
          lang="xml:lang code"
          data-type="text|number|user defined data type" />
```

Position

This element must be a child element of the `xsl:apply-templates` element or the first child element(s) of an `xsl:for-each` element.

xsl:strip-space

The `xsl:strip-space` element is used to control the way whitespace-only nodes in the source document are handled. It is used in conjunction with the `xsl:strip-space` element.

Attributes

`elements`	Indicates the nodes from which whitespace-only nodes should be stripped. This should be a whitespace-separated list of names, possibly using wildcard characters.

Content

This element is always empty.

Usage

```
<xsl:strip-space elements="list-of-name-tests" />
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

xsl:stylesheet

The `xsl:stylesheet` element is the root element of a stylesheet.

Attributes

`id` (optional)	Name to identify the stylesheet when it is embedded in another XML document.
`version`	Defines the version of XSLT required for the stylesheet. The value should be `1.0` until a new version becomes a recommendation.
`extension-element-prefixes` (optional)	List of namespace prefixes used to identify elements that extend XSLT. This should be a whitespace-separated list of namespace prefixes.
`exclude-result-prefixes` (optional)	List of namespaces (prefixes) that should not be copied to the output document unless the namespace is actually used in the output.

Content

The `xsl:stylesheet` element can contain only top-level elements.

Usage

```
<xsl:stylesheet id="Name"
                version="Number"
                extension-element-prefixes="list-of-prefixes "
                exclude-result-prefixes="list-of-prefixes">
```

Position

The `xsl:stylesheet` element is the root element of a stylesheet.

xsl:template

The `xsl:template` element is used to define a unit of processing to produce output known as a template. A template can be called by name or matched based on a matching pattern.

Attributes

`match` (optional)	The matching pattern defining which nodes are matched by the template.
`name` (optional)	The name of the template. Used when the template is called instead of matched.
`priority` (optional)	Defines the priority of the template. This priority is used to determine which template is invoked if more than one template matches a node. The value needs to be a (positive or negative) number. If a priority is not specified, a priority is calculated for a template.
`mode` (optional)	The mode of the template. When `xsl:apply-templates` is used with a mode, only the templates that have the same `mode` identifier are matched with a node.

Content

This element can contain any elements valid in a template body.

Usage

```
<xsl:template match="Pattern"
              name="QName"
              priority="Number"
              name="QName">
```

Position

This element is a top-level element and therefore can occur only as an immediate child element of the `xsl:stylesheet` element.

`xsl:text`

The `xsl:text` element is used to insert text into the output.

Attributes

`disable-output-escaping` (optional)	Determines whether the value of the element should be output escaped. The valid values are `yes` and `no` (default).

Content

This element can contain only text.

Usage

```
<xsl:text disable-output-escaping="yes|no">text</xsl:text>
```

Position

This element can occur anywhere in a template body.

`xsl:transform`

This element is identical to `xsl:stylesheet`. See the reference on `xsl:stylesheet` for details.

`xsl:value-of`

The `xsl:value-of` element is used to insert the value of an expression into the output.

Attributes

`select`	The expression used to determine the value of the output. If the expression does not yield a string value, the result is converted to a string.
`disable-output-escaping` (optional)	Determines whether the value of the element should be output escaped. The valid values are `yes` and `no` (default).

Content

This element is always empty.

Usage

```
<xsl:value-of select="Expression" disable-output-escaping="yes|no" />
```

Position

This element can occur anywhere in a template body.

`xsl:variable`

The `xsl:variable` element is used to define a variable in a stylesheet (global variable) or a template (local variable).

Attributes

`name`	The name of the variable.
`select` (optional)	Expression used to select the value for the variable. The expression may yield a string, number, Boolean, node-set, or tree fragment.

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element. If the `select` attribute is present, the element should be empty.

Usage

```
<xsl:variable name="QName" select="Expression">
```

Position

This element can occur as a top-level element or at any place in a template body.

`xsl:when`

The `xsl:when` element is used to define an alternative choice in an `xsl:choose` element. The body of the `xsl:when` element is processed if it is the first `xsl:when` element where the expression in the `test` attribute returns `true`. If no `xsl:when` element is processed, the `xsl:otherwise` element is processed (if present).

Attributes

`test`	The expression that is used as a condition to process the body or not. The result of the expression should be a Boolean value (otherwise, the result is converted).

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element.

Usage

```
<xsl:when test="Expression">
```

Position

This element can occur only as a child of an `xsl:choose` element.

`xsl:with-param`

The `xsl:with-param` element is used to set the value of a parameter when calling or invoking a template.

Attributes

`name`	Name of the parameter to set.
`select` (optional)	Expression used to select the value for the parameter. The expression may yield a string, number, Boolean, node-set, or tree fragment.

Content

This element can contain any elements valid in a template body, except for those that can be only immediate children of the `xsl:template` element. If the `select` attribute is present, the element should be empty.

Usage

```
<xsl:with-param name="QName" select="Expression">
```

Position

This element can occur as a child element of an `xsl:apply-templates` or `xsl:call-template` element.

XSLT and XPath Function Reference

This section describes the functions in XSLT (and XPath) in alphabetical order.

`boolean()`

The `boolean()` function converts a value to boolean value.

Arguments

`value`	Value to be converted. This may be a literal value or an expression.

Usage

`boolean(`*`value`*`)`

Result

The result depends on the data type of the argument. Table B.1 shows the results per data type.

TABLE B.1 Conversion to Boolean Specified by Data Type

Data Type of Value	*Result*
Number	`0`→ `false`
	Otherwise→ `true`
String	Empty string→ `false`
	Otherwise→ `true`
Node-set	Empty node-set→ `false`
	Otherwise→ `true`
Tree fragment	Conversion via string: `boolean(string(value))`

`ceiling()`

The `ceiling()` function returns the value of an argument rounded to the nearest integer larger than or equal to the value of the argument.

Arguments

`value`	Value to be rounded. This may be a literal value or an expression. If the value is not a number, it is first converted using the `number()` function.

Usage

`ceiling(`*`value`*`)`

Result

The result is an integer rounded to the nearest integer larger than or equal to the value of the argument.

`concat()`

The `concat()` function is used to join two or more strings.

Arguments

Two or more `value` arguments	Values to be joined. They may be literal values or expressions. Any value that is not a string is first converted using the `string()` function.

Usage

```
concat(value1, value2, value3, ...)
```

Result

A string consisting of all values passed joined together.

contains()

The `contains()` function is used to check whether a string contains a certain sequence of characters.

Arguments

`value`	String value to be searched.
`string`	String searched for.

Usage

```
contains(value, string)
```

Result

If the given value contains the given string, the function returns `true`; otherwise, it returns `false`.

count()

The `count()` function counts the number of nodes in a node-set.

Arguments

`node-set`	Expression returning the node-set to be counted.

Usage

```
count(node-set)
```

Result

The result is an integer with the number of nodes in the node-set given as an argument.

`current()`

The `current()` function returns the current node. This function is used in expressions where the context of the expression is not the current node and the current node is needed in the expression.

Arguments

This function has no arguments.

Usage

`current()`

Result

The result is a node-set containing one node: the current node.

`document()`

The `document()` function is used to get access to an XML source document other than the document being processed.

Arguments

`URI`	Either a value that evaluates to a string with one URI, or a node-set with multiple URIs. In the latter case, all documents specified by the URIs are accessible.
`base-uri` (optional)	The base-URI is used as the base-URI for any relative URI in the `URI` argument. The value needs to be a node-set, of which the first node is used as the base-URI.

Usage

document(*URI*)

document(*URI*, *base-URI*)

Result

A node-set created from the XML tree in the specified documents. `document('')` gives access to the elements in the stylesheet processing the source XML.

`element-available()`

The `element-available()` function checks whether the processor supports a certain XSLT or XSLT extension element.

Arguments

`name`	Name of the element to check for, including its namespace prefix.

Usage

`element-available(`*`name`*`)`

Result

If the element is available, the function returns `true`; otherwise, it returns `false`.

`false()`

The `false()` function returns the Boolean corresponding to `false`. This function is required because there is no actual value to compare an expression to. The `false()` function performs this task.

Arguments

This function has no arguments.

Usage

`false()`

Result

The result is always the Boolean value `false`.

`floor()`

The `floor()` function returns the value of an argument rounded to the nearest integer smaller than or equal to the value of the argument.

Arguments

`value`	Value to be rounded. This may be a literal value or an expression. If the value is not a number, it is first converted using the `number()` function.

Usage

`floor(`*`value`*`)`

Result

The result is an integer rounded to the nearest integer smaller than or equal to the value of the argument.

format-number()

The `format-number()` function is used to format a number value for output. The result depends on the decimal format that is used, as defined by the `xsl:decimal-format` element.

Arguments

`value`	Value to be formatted. If the given value is not a number, it is first converted using the `number()` function.
`format`	The pattern according to which the number is to be formatted.
`name` (optional)	The name of the decimal format to be used, as defined with `xsl:decimal-format`.

Usage

`format-number(`*`value`*`, `*`format`*`)`

`format-number(`*`value`*`, `*`format`*`, `*`name`*`)`

Result

The function returns a string value with the number value formatted according to the pattern and optional decimal format.

function-available()

The `function-available()` function checks whether the processor supports a certain XSLT or XSLT extension function.

Arguments

`name`	Name of the function to check for, including its namespace prefix if applicable.

Usage

`function-available(`*`name`*`)`

Result

If the function checked for is available, the function returns `true`; otherwise, it returns `false`.

generate-id()

The `generate-id()` function is used to generate a string that uniquely identifies a node.

Arguments

`node` (optional)	Node to generate the unique identifier for. The value is actually a node-set, of which the first node is evaluated. If this attribute is omitted, the context node is used.

Usage

`generate-id()`

`generate-id(`*`node`*`)`

Result

The function returns a string with the unique identifier.

`id()`

The `id()` function is used to access nodes through their ID attribute, which uniquely identifies a node among nodes of the same type.

Arguments

`value`	Value of the ID attribute of the node(s) to be found. The value depends on the data type of the ID attribute.

Usage

`id(`*`value`*`)`

Result

The function returns a node-set with nodes that have an ID attribute value corresponding to the given value.

`key()`

The `key()` function is used to find the nodes that have a given value for a named key. This function is used in conjunction with the `xsl:key` element.

Arguments

`name`	Name of the key.
`value`	Value of the key.

Usage

`key(`*`name`*`, `*`value`*`)`

Result

The function returns a node-set with all the nodes that have the given value for the given key.

`lang()`

The `lang()` function is used to check the language of the context node, as defined by an `xml:lang` attribute.

Arguments

`language`	Language to check for. This should be a `valid xml:lang` code.

Usage

`lang(`*`language`*`)`

Result

The function returns `true` if the language of the context node is equal to the language code given as the argument, or if the language of the context node is a sub-language of the language given by the language code. Otherwise, the function returns `false`.

`last()`

The `last()` function returns the position number of the context node. This function is useful only when you're processing a list of nodes.

Arguments

The function has no arguments.

Usage

`last()`

Result

The result is a number corresponding to the position in the list of nodes the context node is part of.

`local-name()`

The `local-name()` function returns the name of an element, without the namespace prefix (if present).

Arguments

`node` (optional)	The node whose local name should be given. If this attribute is omitted, the local name of the context node is returned.

Usage

`local-name()`

`local-name(`*`node`*`)`

Result

The function returns a string containing the local name.

`name()`

The `name()` function returns the name of an element, including the namespace prefix (if present).

Arguments

`node` (optional)	The node whose name should be given. If this attribute is omitted, the name of the context node is returned.

Usage

`name()`

`name(`*`node`*`)`

Result

The function returns a string containing the name.

`namespace-uri()`

The `namespace-uri()` function returns the URI associated with the namespace of the context element.

Arguments

`node` (optional)	The node whose namespace URI should be given. If this attribute is omitted, the namespace URI of the context node is returned.

Usage

`namespace-uri()`

`namespace-uri(`*`node`*`)`

Result

The function returns a string containing the namespace URI.

`normalize-space()`

The `normalize-space()` function removes all leading and trailing whitespace in a string and replaces all sequences of whitespace characters in the string with a single space. This is similar to how HTML deals with whitespace.

Arguments

`value` (optional)	The string value to be normalized. If this attribute is omitted, the string value of the context node is normalized.

Usage

```
normalize-space()
```

```
normalize-space(value)
```

Result

The function returns a normalized string. A normalized string is a string with all leading and trailing whitespace removed, and all whitespace in the value replaced by a single space.

`not()`

The `not()` function is used to get the opposite Boolean value of the argument.

Arguments

`value`	The value to be negated. The value may be an expression. If the value is not a Boolean value, it is first converted using the `boolean()` function.

Usage

```
not(value)
```

Result

The result is a Boolean value that has the negated value of the given value.

`number()`

The `number()` function converts a value to a number value.

Arguments

`value` (optional)	Value to be converted. This may be a literal value or an expression. If this attribute is omitted, the string value of the context node is converted.

Usage

`number()`

`number(`*`value`*`)`

Result

The result depends on the data type of the argument. Table B.2 shows the results per data type.

TABLE B.2 Conversion to Number Specified by Data Type

Data Type	*Result*
Boolean	`false`→ `0`
	`true`→ `1`
String	Parsed as a decimal number
Node-set	Conversion via string
Tree fragment	Conversion via string

`position()`

The `position()` function returns the ordinal position of the context node, within a list of nodes.

Arguments

This function has no arguments.

Usage

`position()`

Result

The result is a number with the ordinal position of the context node.

`round()`

The `round()` function rounds a number according to regular mathematical rules.

Arguments

`value`	Value to be rounded. If the value is not a number, it is first converted using the `number()` function.

Usage

`round(`*`value`*`)`

Result

The function returns an integer that is the result of rounding the argument following regular mathematical rules.

`starts-with()`

The `starts-with()` function checks whether a string starts with a certain sequence of characters.

Arguments

`value`	String value to be searched.
`string`	String searched for.

Usage

`starts-with(`*`value`*`, `*`string`*`)`

Result

If the given value starts with the given string, the function returns `true`; otherwise, it returns `false`.

`string()`

The `string()` function converts a value to a string.

Arguments

`value` (optional)	Value to be converted. This may be a literal value or an expression. If this attribute is omitted, the string value of the context node is converted.

Usage

`string()`

`string(`*`value`*`)`

Result

The result depends on the data type of the argument. Table B.3 shows the results per data type.

TABLE B.3 Conversion to String Specified by Data Type

Data Type	*Result*
Boolean	`false`→ `'false'`
	`true`→ `'true'`
Number	Converted to a string in decimal number format
Node-set	String value of the first node in the node-set (in document order)
Tree fragment	Concatenate all string values in the tree fragment

`string-length()`

The `string-length()` function returns the length of a given string.

Arguments

`value` (optional)	Value of which the string length is required. If the value is not a string, it is first converted using the `string()` function. If this attribute is omitted, the string length of the string value of the context node is returned.

Usage

`string-length()`

`string-length(`*`value`*`)`

Result

The function returns a number with the length of the given string.

`substring()`

The `substring()` function returns part of a string based on the position of the characters in the string.

Arguments

`value`	Value from which a substring needs to be taken. If the value is not a string, it is first converted using the `string()` function.
`start`	Position at which the substring starts within the given value. Characters are counted from 1. If the given start value is not a number, it is first converted using the `number()` function.

`length` (optional)	The number of characters to be included in the substring, starting at the starting point. If the given length value is not a number, it is first converted using the `number()` function. If the argument is omitted, the substring that is returned starts at the starting point and includes the remainder of the string.

Usage

substring(*value*, *start*)

substring(*value*, *start*, *length*)

Result

The function returns a string that contains part of the string given as the value.

substring-after()

The `substring-after()` function returns a substring that occurs after a given sequence of characters.

Arguments

`value`	Value from which a substring needs to be taken. If the value is not a string, it is first converted using the `string()` function.
`substring`	Substring that needs to occur in the given value. If the value is not a string, it is first converted using the `string()` function.

Usage

substring-after(*value*, *substring*)

Result

The function returns a substring of the given string that occurs after the first occurrence of the given substring in the given string.

substring-before()

The `substring-before()` function returns a substring that occurs before a given sequence of characters.

Arguments

`value`	Value from which a substring needs to be taken. If the value is not a string, it is first converted using the `string()` function.
`substring`	Substring that needs to occur in the given value. If the value is not a string, it is first converted using the `string()` function.

Usage

`substring-before(`*`value`*`, `*`substring`*`)`

Result

The function returns a substring of the given string that occurs before the first occurrence of the given substring in the given string.

`sum()`

The `sum()` function returns the sum of the number values of a node-set.

Arguments

`node-set`	The nodes whose number values need to be added together. This should be a node-set and will yield an error if it is not.

Usage

`sum(`*`node-set`*`)`

Result

The function returns a number with the summed value of the node-set.

`system-property()`

The `system-property()` function returns information about the processor.

Arguments

`property`	The name of the system-property to be returned. Valid values are `xsl:version` (XSLT version supported by the processor), `xsl:vendor` (the name of the vendor) or `xsl:vendor-url` (the URL of the vendor's Web site). If the value is not a string, it is first converted using the `string()` function.

Usage

`system-property(`*`property`*`)`

Result

The function returns a string with the requested value.

`translate()`

The `translate()` function is used to substitute characters in a string.

Arguments

`value`	The string in which characters should be substituted. If the value is not a string, it is first converted using the `string()` function.
`from`	List of characters to be replaced. If the value is not a string, it is first converted using the `string()` function.
`to`	List of substitute characters. If the value is not a string, it is first converted using the `string()` function.

Usage

```
translate(value, from, to)
```

Result

The function returns the given string `value`, with the characters specified in the `from` argument substituted by those in the `to` argument.

true()

The `true()` function returns the Boolean corresponding to `true`. This function is required because there is no actual value to compare an expression to. The `true()` function performs this task.

Arguments

This function has no arguments.

Usage

```
true()
```

Result

The result is always the Boolean value `true`.

unparsed-entity-uri()

The `unparsed-entity-uri()` function gives access to unparsed entity declarations in the DTD of the source document.

Arguments

`name`	The name of the unparsed entity to be returned. If the value is not a string, it is first converted using the `string()` function.

Usage

`unparsed-entity-uri(`*`name`*`)`

Result

The function returns a string with the URI (system identifier) of the unparsed entity of the given name. If the unparsed entity does not exist, an empty string is returned.

APPENDIX C

Command-Line Reference for Selected Processors

In this book, you have learned about several processors. These processors have been used with only their most basic options. This reference shows you all the options for the different processors and tells you the effect of each option.

The processors discussed are

- MSXSL
- Saxon
- Xalan

If you want to use another processor, check out `http://xsl.startkabel.nl/`, which contains links to most known processors.

MSXSL

Microsoft has a parser/processor component named MSXML, which works on any version of Windows 95 or higher, or Windows NT 4.0 or higher. Several versions are in circulation, so for proper XSLT support, you need to make sure that your system contains MSXML 3.0 or higher. Installing Internet Explorer 6.0 or higher will do the trick. Otherwise, you can download the latest version from `http://msdn.microsoft.com/xml`.

You can't run the component itself from the command line; only developers creating applications with Visual Basic, Visual C++, ASP, and so on can use it. MSXSL is a utility that lets you use the MSXML parser/processor component from the command line. This utility needs to be downloaded and installed separately. MSXSL can also be downloaded from `http://msdn.microsoft.com/xml`.

Usage

When you invoke MSXSL from the command line, be sure that you type `MSXSL`, the name of the utility, and not MSXML, the name of the component. You can run MSXSL from the command line as follows:

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

Any number of *`param=value`* pairs can be added to the command line, to pass parameters to the stylesheet.

The parameters may use a namespace prefix that is not defined in the stylesheet. To resolve this problem, you can add any number of namespace prefixes with their URIs. This is also valid for a start mode with a namespace prefix (see Table C.1).

Options

Table C.1 shows a list of command-line options for MSXSL, along with their meaning.

TABLE C.1 Options for MSXSL

Option	*Description*
`-?`	Shows a message about the usage of MSXSL.
`-o` *`filename`*	Writes the output to the named file.
`-m` *`startmode`*	Starts the transformation in the specified mode.
`-xw`	Strips nonsignificant whitespace from the source and stylesheet.
`-xe`	By default, MSXSL resolves external references such as DTD external subsets or external entity references in the XML source and the stylesheet. This option disables this behavior. This does not have any effect on documents included, imported, or accessed through the `document()`function.
`-v`	Validates documents during the parse phase.
`-t`	Shows the time it took to load and transform the source document.
`-`	A dash in place of the source document uses stdin as a source document.
`-`	A dash in place of the stylesheet uses stdin as a stylesheet.

Saxon

Saxon is a processor developed by Michael Kay, one of the leading XSLT experts. Although it is not a corporate product, it is one of the best processors around, specifically where standards compliance is concerned. You can download the latest version of Saxon from `http://users.iclway.co.uk/mhkay/saxon/`.

There are two versions of Saxon. Full Saxon is the full product with all documentation, source code, Java executables, and samples. Windows users can also use Instant Saxon, which is a Windows executable that can be run from the command line.

Usage

You can use Instant Saxon as follows:

```
SAXON [options] source stylesheet [param=value...]
```

Full Saxon runs under Java, so you need to invoke Java with the correct class name, as follows:

```
java com.icl.saxon.StyleSheet [options] source stylesheet [param=value...]
```

Any number of *`param=value`* pairs can be added to the command line, to pass parameters to the stylesheet.

Options

Table C.2 shows a list of command-line options for Saxon, along with their meaning.

TABLE C.2 Options for Saxon

Option	*Description*
`-?`	Shows a message about the usage of Saxon.
`-a`	Uses the stylesheet linked to the source XML instead of a stylesheet specified on the command line.
`-ds`	Selects the implementation of the internal tree model and sets it to the traditional model.
`-dt`	Selects the implementation of the internal tree model and sets it to the “tinytree” model (default).
`-o` *`filename`*	Writes the output to the named file
`-m` *`classname`*	Uses the specified `Emitter` to process the output from `xsl:message`. The class must implement the `com.icl.saxon.output.Emitter` class. By default, the standard XML emitter is used; it writes to the standard error stream.

Table C.2 Continued

Option	*Description*
`-r` *`classname`*	Uses the specified `URIResolver` to process all URIs. The `URIResolver` is a user-defined class that must extend the `com.icl.saxon.URIResolver` class.
`-t`	Shows the time it took to load and transform the source document, and version information about the processor and the Java version.
`-T`	Shows stylesheet tracing information useful for debugging. The tracing information is sent to the standard (error) output and contains line numbers of the stylesheet. The tracing information is in an XML format.
`-TL` *`classname`*	Uses the specified `TraceListener` when processing the stylesheet. The *`classname`* names a user-defined class that must implement `com.icl.saxon.trace.TraceListener`.
`-u`	Specifies that the filenames given for the XML source and the stylesheet are URLs instead of local filenames.
`-w0`	Specifies that Saxon should not display error messages for recoverable errors.
`-w1`	Specifies that Saxon should display error messages for recoverable errors and continue processing (default setting).
`-w2`	Specifies that Saxon should treat recoverable errors as fatal errors and stop processing.
`-x` *`classname`*	Specifies that Saxon should use the specified SAX parser for the source file and any file loaded with the `document()` function. The specified parser must be the fully qualified class name of a Java class that implements the `org.xml.sax.Parser` or `org.xml.sax.XMLReader` interface.
`-y` *`classname`*	Specifies that Saxon should use the specified SAX parser for the stylesheet and any stylesheets included or imported into it. The specified parser must be the fully qualified class name of a Java class that implements the `org.xml.sax.Parser` or `org.xml.sax.XMLReader` interface.

Xalan

Xalan, a popular processor, was developed by the Apache XML Project (`http://xml.apache.org/`). Two versions of Xalan are available. Xalan C++ is the first version that was implemented in C++. The current version is Xalan Java 2, which is an entirely new implementation in Java. In this book, I used *Xalan* to indicate *Xalan Java 2*.

Usage

When you run Xalan, remember that it runs under Java, so you need to invoke Java with the correct class name, as follows:

```
java org.apache.xalan.xslt.Process -IN source -XSL stylesheet [options]
```

You can add any number of parameters on the command line to be used in the stylesheet. Unlike in the other processors, you have to specify each parameter as an option (see Table C.3).

Options

Table C.3 shows a list of command-line options for Xalan, along with their meaning.

TABLE C.3 Options for Xalan

Option	*Description*
`-OUT` *`filename`*	Writes the output to the named file.
`-V`	Displays Xalan version information.
`-QC`	(Quiet Pattern Conflicts Warnings)
`-Q`	(Quiet Mode)
`-LF`	Outputs linefeed (LF) characters only instead of carriage return (CR) and linefeed (LF).
`-CR`	Outputs carriage return (CR) characters only instead of carriage return (CR) and linefeed (LF).
`-INDENT` *`number`*	Indents each level in the output tree with the specified number of spaces. The default number is 0.
`-TT`	Displays tracing information about the templates being called or matched. Gives information on the line number and file containing the template.
`-TG`	Displays tracing information about the result tree generation. Shows when a start or end tag is inserted and when text is inserted.
`-TS`	Displays tracing information for each `select` event in the stylesheet.
`-TTC`	Displays tracing information about the processing of child elements of template elements.
`-EDUMP` *`filename`*	When an error occurs, the stack is dumped to file. The filename is optional, and if not specified, the stack is dumped to stdout.
`-XML`	Creates XML output. This is the same as `<xsl:output method="xml" />` in the stylesheet.

TABLE C.3 Continued

Option	*Description*
`-TEXT`	Creates text output. This is the same as `<xsl:output method="text" />` in the stylesheet.
`-HTML`	Creates HTML output. This is the same as `<xsl:output method="html" />` in the stylesheet.
`-PARAM` *`name value`*	Adds a stylesheet parameter with the given name and given value. For multiple parameters, add this option multiple times.
`-DIAG`	Shows the time it took to load and transform the source document.
`-URIRESOLVER` *`classname`*	Uses the specified `URIResolver` to process all URIs. The `URIResolver` is a user-defined class that must implement the `javax.xml.transform.URIResolver` interface.
`-ENTITYRESOLVER` *`classname`*	Uses the specified `EntityResolver` to process all entities. The `EntityResolver` is a user-defined class that must implement the `org.xml.sax.EntityResolver` interface.
`-CONTENTHANDLER` *`classname`*	Uses the specified `ContentHandler`. The `ContentHandler` is a user-defined class that must implement the `org.xml.sax.ContentHandler` interface.

APPENDIX D

XML Resources on the Web

The Extensible Markup Language (XML) is a standard that was designed with the Web in mind. It is not surprising, therefore, that you can find quite a few good resources on the Web. The following list describes the best resources out there. This list is by no means complete, but it provides you with some good starting points. Most of these sites are not XSLT sites by themselves but offer information on XML, XSLT, and related technologies.

- `http://xml.apache.org/`—This site contains all the details about the Apache XML Project, an Open Source project creating many useful tools for XML developers working with Java.
- `http://www.aspfriends.com/aspfriends/xml.asp`—If you're a Web developer using XML or XSLT with ASP or ASP.NET, this site is definitely the place to go for questions. Many of the industry experts are on these mailing lists.
- `http://www.ibm.com/developerworks/xml/`—This extensive XML resource by IBM provides a great deal of information about XML on different platforms.

- `http://msdn.microsoft.com/xml/`—This section of the Microsoft Developer Network site is only about Microsoft XML technologies. The site contains downloads of the latest parsers/processors and many articles and references you'll find useful if you're a Microsoft-oriented developer.
- `http://technet.oracle.com/tech/xml/`—This Oracle developer XML site offers articles and information on XML, mostly aimed at use with Oracle products, of course.
- `http://www.perfectxml.com/`—This site, which provides articles, book chapters, and software, is one of the most comprehensive sites on XML out there.
- `http://xsl.startkabel.nl/`—Startkabel is a collection of portals. The comprehensive XSL portal at Startkabel provides links to tutorials, mailing lists, editors, tools, and most XSLT processors known to man.
- `http://www.sun.com/xml/`—Sun's XML section focuses on XML in Sun technology. This is a site you can't do without if you're a Java developer working with XML and XSLT.
- `http://www.topxml.com/`—TopXML is full of articles, tutorials, and references. One nice thing about this site is that visitors can post handy code snippets. This feature has made it a real community site. You can also subscribe to several mailing lists such as XSLTalk, a mailing list where you can post your XSLT questions.
- `http://www.w3.org/`—The site of the W3 Consortium is responsible for the standards on the Web. This site contains the actual specifications for XML, XSLT, XML Schemas, XPath, SOAP, and so on. It also contains links to other resources and information about new books and applications in the XML field.
- `http://www.xml.com/`—This XML site provides articles and tutorials, which offer good startup material on many topics.
- `http://www.xml.org/`—XML.org is an XML portal sponsored by some of the industry leaders, such as IBM and Oracle. It features articles, newsletters, and links. It also is the home of the XML-DEV mailing list, one of the key XML mailing lists out there.
- `http://xml.coverpages.org/`—This site doesn't look all that great, but then looks can be deceiving. It features information on XML, XSL, and CSS, but also on older standards such as DSSSL.
- `http://www.sys-con.com/xml/`—The XML Journal offers a wide range of articles on XML and related technologies.
- `http://www.xslt.com/`—XSLT.com is only about XSLT, not other XML technologies like most other sites. The site offers a handy grouping of links to other resources and tutorials, and links to XSLT editors, parsers/processors, tools, and utilities.

INDEX

Symbols

8-bit integers, 251
16-bit integers, 251
32-bit integers, 251
(@) character, 75
(|) character, 92
(=) equal sign operator, 150
(//) expression, 79
(/) forward slash, 154
(&gt) greater than sign operator, 150
(&lt) less than sign operator, 150
(!=) not equal sign operator, 150
(*) wildcard character (XPath), 72-73
  predicates, 74

A

absolute addressing, 60
accessing
  stylesheet elements, 357-359
  variables, 211
  XML sources, 342
ad hoc data, 91-92
adding
  data to parameters, 242-244
  priorities to templates, 103-104
  templates, 43-44
advantages
  keys, 405
  namespaces, 380-381
    languages, mixing, 380
  parameters, 231-232
  templates, 86
    ad hoc data, 91-92
    multiple documents with one stylesheet, 86-91
  variables, 209
    performance, 209-210
    values, 210
  XSLT, 16
  XSLT extensions, 466-467
aliases, namespaces, 389-392
and operator, 157
Apache XML Project Web site, 31
API (Application Programming Interface), 15

Application Programming Interface (API), 15
applications, computational, 451-458
apply-templates element (XSL), 184
arguments
 contains() function, 273
 xsl:vendor, 490
 xsl:vendor-url, 490
 xsl:version, 490
attribute axis, 75
attribute-sets, 128-129, 335-337
 border, 128
 tableprop, 128-129
 width, 128
 XSLT, 522
attributes, 16
 case-order, 299-300
 currency, 461
 default pattern characters, 289
 elements, comparing, 61
 exclude-result-prefixes, 392, 489
 extension, 466
 extension-element-prefixes, 469
 href, 321
 ID, 413
  keys, 418-419
  naming, 414
  unique, 414-418
 lang, 299-300
  valid values, 300-301
 match, 40-43, 100
 media, 52
 mode, 98
  templates, 98-100
 name, 96
 order, 298, 488
 output element
  encoding, 175
  media-type, 174
 priority, 103
 select, 296
 templates, 113
  attribute-sets, 128-129
  inserting, 123-124
  inserting literal attributes, 127-128
  inserting with generated names, 124-127
 use-attribute-sets, 128
 values, 70-72
  troubleshooting, 488
 version, 36-37
 XML, 514-515
 XML document trees, 60
 xmlns, 378
axis, 75-76
 attribute, 75
 multilevel, 77-80
 parent, 75

B

Base URIs, 357
benefits. ***See*** **advantages**
binary data, 171-173
BizTalk, 20
BizTalk Web site, 20
Boolean data types, 252-253
 conversion to, 260-261
Boolean functions, 157-158
 expressions results, 158
  conversion to Boolean values, 159-160
boolean() function, 259
border attribute-set, 128
built-in XSLT extensions, 468
 Saxon extension, 468-469
  elements, 469-472
  functions, 473-475

C

calling
 processors, 245-246
 XSLT stylesheets, 521-522
Cascading Stylesheets. ***See*** **CSS**
case-order attributes, 299-300
 valid values, 300
CDATA sections, 176-179
character encoding, 487-488
characters
 entities, inserting into templates, 108-113
 output-escaped, 113

substrings, 281-283
after characters, 281-283
before characters, 281-283
text, inserting into templates, 108-110

child elements, 61

child nodes (XPath), 66-67

code
parameters, 244, 246
stylesheets in XML sources, 54-55

colors, variables, 212-214

combining
expressions, 156-157
match and named templates, 98

command-line options
MSXSL, 618
Saxon, 619-620
Xalan, 621-622

comments, inserting into XML documents, 133-135

communicating, XML and XSLT, 19-21

comparing
attributes and elements, 61
data, expressions, 400-402
importing and including stylesheets, 329-330
match and named templates, 96-98
parameters and variables, 230
processors, 487
values, 263-264
conversion rules, 263
disadvantages, 264

comparison operators, 150

complex variables, 215-216

composite numbering, 312-314

computational applications, 22, 451-458

computational stylesheets, 442
functions, 446
mathematical, 446-451
operators, 444-446
usage, 443

concat() function, 271

conditional processing, 148
comparison operators, 150
multiple options, 152-154
xsl:if element, 148-151

contains() function, 272-273

content, xsl:stylesheet element, 39-40

context node, 70, 129-131

controlling output, 163-164
binary data, 171-173
encoding types
determining, 174-175
unsupported character sets, 175-176
escaped characters, 176-179
HTML output
creating, 167-169
example, 169-171
versions, 169
media types, 174
PostScript output, 171
processor extensions, 173-174
text output method, 171
text-based formatting, 171
whitespace, 179
deleting, 180-185
example, 179-180
indented code, 188-189
preserving, 181-183
stylesheets, 185-188
XML output, 164-165
converting to PDF, 173-174
external general parsed entities, 164
forcing, 165-166
versions, 166-167
well-formed XML, 164
XML declarations, 167

conversions
data types, 259
disadvantages, 263
implicit, 262
to Boolean data types, 260-261

to number data types, 261-262
to string data types, 262
values, comparing, 263
converting
currencies, 460
XML output to PDF, 173-174
copying elements from source documents, 129, 131-133
context node, 129-131
node-sets, 131-133
tree fragments, 131-133
Corning (Michael), 518
creating
expressions to match nodes, 92-96
HTML output, 167-169
parameters, 231
variables, 210-215
data sets, 220-224
expressions, 216-220
iterations, 220-224
multiple, 216
nodes, 225-226
templates, 220-224
XSLT elements, 225
XSLT files, 27
Text Editor, 27
XML editors, 27-28
XSLT debuggers, 28-29
XSLT design tools, 29
XSLT editors, 28-29
CSS (Cascading Stylesheets), 22, 36
stylesheet types, 36
currencies, converting, 460
currency attribute, 461
current element, 61
current nodes, 61-62

D

data
expressions
comparing, 400-402
matching, 400-401
selecting, 400-401
formatting, 285
keys, 405-412
parameters, 231, 240
adding data to, 242-244
sorting, 296
storing, 250
variables, creating for, 220-224
XML
Oracle, 498-502
SQL servers, 502-503
XML sources, 45, 342-345
combining data, 347-349
matching data, 349-352
pulling data, 350
storing data in variables, 345-347
xsl:value-of element, 45-47
data storage, 16-19
data types, 249-250
16-bit integers, 251
32-bit integers, 251
8-bit integers, 251
Boolean, 252-253
conversions, 259
disadvantages, 263
implicit, 262
to Boolean data types, 260-261
to number data types, 261-262
to string data types, 262
defined, 250
double precision, 251
dynamically typed languages, 252
floating-point numbers, 251
integers, 251
node-set, 257-259
number, 253
single precision, 251
sorting, 301
string, 256-257
tree fragments, 258
XML Schemas, 252
XSLT, 251-252
data warehousing applications, 22
databases, servers (XML), 497
date and time, formatting, 290-291
decimal formats, 337
decimal formatting patterns, 286

declarations
- namespaces, 489
- removing from output, 167

declarative programming languages, 26-27
declaring namespaces, 378
deep copy, 131
default number format, 289
default pattern characters, 289
default priorities, templates, 101-103
defining XML sources dynamically, 353-355
definitions
- data types, 250
- parameters, 230
- recursion, 424
- stylesheets, 36
- variables, 208-209

deleting
- whitespace
  - from source documents, 180-181
  - from values, 183-184
  - potential problems, 184-185
- XML declarations from output, 167

descending order, sorting, 298-299
descriptors, media, 52
designing
- XML, 508-510
  - attributes, 514-515
  - elements, 514-515
  - hierarchy, 510-514
  - multiple documents, 515-516
  - namespaces, 516-518
  - tools, 518
  - XML Schemas, 518
- XSLT, 519
  - attribute-sets, 522
  - calling, 521
  - iteration, 521
  - matching, 521-522
  - multifile stylesheets, 523
  - rules, 524-525
  - stylesheets, 519-520
  - troubleshooting errors, 523-524
  - variables, 522

disabling output escaping, 176-179
disadvantages. *See* limitations
div operators, 444
Document Style Semantics and Specification Language (DSSSL), 12
Document Type Definitions. *See* DTDs
document() function, 342-347
- data, storing in variables, 345-347
- file locations, 357
- files, opening, 354
- stylesheet elements, accessing, 357-359

documents
- whitespace, 179
  - deleting, 180-185
  - example, 179-180
  - indented code, 188-189
  - preserving, 181-183
  - stylesheets, 185-188
- XML sources, linking, 355-357

DOM, API, 15
double precision, 251
DSSSL (Document Style Semantics and Specification Language), 12
DTDs (Document Type Definitions), 13, 413
- ID attributes, 413
  - naming, 414
- internal, 413-414
- namespaces, 382-383

dynamic elements, 119-121
dynamic sorting, 296, 304-306
dynamically typed languages, 252

E

EDI (Electronic Data Interchange), 21
Electronic Data Interchange (EDI), 21
element-available() function, 492-494

elements, 16
attributes, comparing, 61
child, 61
copying from source documents, 129, 131-133
context node, 129-131
node-sets, 131-133
tree fragments, 131-133
current, 61
dynamic, 119-121
extension, 466
Saxon, 469-472
including stylesheets, 327-328
msxsl:script, 480
node-sets, 68
parent, 311
predicates, 69
processors, 492-493
root, 14
saxon:assign, 470-472
saxon:sum(), 473-475
saxon-assign, 497
stylesheets, 37-38
namespaces, 37-38
xsl:stylesheet element, 39-40
xsl:transform element, 38
templates, 85, 113-116
dynamic elements, 119-121
generated names, 118-119
inserting into output, 114
name() function, 121-123
tags, 116
XML structure, 116-118
values, 67-70
XML, 514-515
XML document trees, 60
absolute addressing, 60
relative addressing, 60
xsl:apply-imports, 333
xsl:apply-templates, 43-45, 301-304
xsl:call-template, 428-430
xsl:choose, 152
xsl:copy, 129, 131
xsl:copy-of, 132-133
xsl:decimal-format, 288-289, 337
xsl:fallback, 496
xsl:for-each, 142, 296-298
xsl:if, 148-151
xsl:import, 330
xsl:include, 321-322, 488
href attribute, 321
xsl:key, 405-412
xsl:namespace-alias, 390-392
xsl:number, 306-310
xsl:otherwise, 152
xsl:output, 337, 488
xsl:preserve-space, 337, 339
xsl:sort, 296
order attribute, 298
select attribute, 296
xsl:strip-space, 337, 339
xsl:stylesheet, 37, 153, 519
content, 39-40
version attribute, 37
xsl:template, 40, 153
xsl:transform, 38
xsl:value-of, 45-47
xsl:when, 152
xsl.element, 118
XSLT
variables, creating from, 225-226
versions, 495-497
XSLT stylesheets, 36
elements (XSL)
apply-templates, 184
output, 165-166
encoding attribute, 175
example, 166
media-type attribute, 174
preserve-space, 182-183
strip-space, 180-181
text, 186-187
embedding stylesheets in XML sources, 52, 54
empty strings, 279
encoding, character, 487-488
encoding attribute (output element), 175

encoding types
determining, 174-175
unsupported character sets, 175-176
entities, 108-109
HTML, 110-113
equal sign (=), 150
escaping, disabling, 176-179
eXcelon Stylus Studio, 28
eXcelon Stylus Studio Web site, 28
exclude-result-prefixes attribute, 392, 489
expressions, 400
(//), 79
Boolean functions, 158
conversion to Boolean values, 159-160
data
comparing, 400-402
matching, 400-401
selecting, 400-401
formalizing, 154-155
combined expressions, 156-157
predicates, 156
values, 155
values, selecting, 402-404
variables, creating, 216-220
XPath, 64-65
child nodes, 66-67
Extensible Markup Language. *See* XML
Extensible Stylesheet Language Transformations. *See* XSLT
extension attributes, 466
extension elements, 466
extension functions, 466
extension-element-prefixes attribute, 469
extensions
Java
functions, 476-478
functions with scripts, 478-482
output formats, 173-174
processors, 475
Saxon, 468-469
elements, 469-472
functions, 473-475
XSLT, 466
advantages, 466-467
built-in, 468-469
elements, 469-472
extension attributes, 466
extension elements, 466
extension functions, 466
functions, 473-475
limitations, 467-468
usage, 468
external general parsed entities, 164

F

factoral function, 429-430
false() function, 157
file references, XML sources, 353-355
files
multiple, 320
limitations, 320-321
specifying with node-set, 355
XML sources, locations, 357
XSLT, 27
Text Editor, 27
XML editors, 27-28
XSLT debuggers, 28-29
XSLT design tools, 29
XSLT editors, 28-29
filtering, node-sets, 144-146
finite non-zero values, 254
firing, templates, 44-45
floating-point numbers, 251
FOP utility, 172
forcing XML output, 165-166
formalizing expressions, 154-155
combined expressions, 156-157
predicates, 156
values, 155

format-number() function, 285, 290
localization, 287-288
formatting
data, 285
date and time, 290-291
numbers, 285-286
decimal formatting patterns, 286
default pattern characters, 289
negative numbers, 287
positive numbers, 287
prefixes, 287
special number values, 288
suffixes, 287
forward slash (/), 154
Full Saxon, 619
function-available() function, 494
functions
Boolean, 157-158
expression results, 158-160
boolean(), 160, 259
computational stylesheets, 446
mathematical, 446-451
concat(), 271
contains(), 272-273
arguments, 273
document(), 342-347
storing in variables, 345-347
file locations, 357
files
opening, 354
accessing, 357, 359
element-available(), 492, 494
extension, 466
Java, 476-482
Saxon, 473-475
factoral, 429-430
false(), 157
format-number(), 285, 290
localization, 287-288
function-available(), 494
generate-id(), 415-419
id(), 414
key(), 405-412, 419
last(), 147
local-name(), 394, 396
max, 239-240
namespace-uri(), 386
node-sets, 146-148
normalize-space, 183
not(), 158
number(), 259
position(), 147, 309
processors, 494-495
recursion, 424-425
saxon:expression(), 474
start-with(), 274-277
string, 458-460
string(), 259
string-after(), 283-285
string-before(), 283-285
string-length(), 278-279
substring(), 281
substring-after(), 281, 283, 394
substring-before(), 281, 283
sum(), 447-450
system-property(), 490-491
xsl:vendor argument, 490
xsl:vendor-url argument, 490
xsl:version argument, 490
templates, creating, 236-239
text(), 47
true(), 157
XSLT, 239-240

G

Generalized Markup Language (GML), 12
generate-id() function, 415-419
global parameters, 230
GML (Generalized Markup Language), 12
greater than sign (&gt), 150
grouping, numbering, 316

H

hierarchy, XML, 510-514
href attribute, 321
HTML (Hypertext Markup Language),
creating, 167-169
example, 169-171

including stylesheets, 323-327
syntax, 48
HTML documents, exclude-result-prefixes attribute, 392
HTML entities, 110-113

I

ID attributes, 413
ID, 415
IDREFS, 415
keys, values, 418-419
naming, 414
unique, 414
inserting, 415-418
linking with elements, 415-418
id() function, 414
IDREFS, ID attributes, 415
implicit data type, conversions, 262
importing stylesheets, 329-331
attribute-sets, 335-337
decimal formats, 337
output, 337
overriding templates, 332-335
parameters, 339
variables, 339
whitespace, 337-339
including stylesheets, 321
comparing with importing, 329-330
duplicate parameters, 329
duplicate templates, 328-329
duplicate variables, 329
elements, 327-328
HTML code, 323-327
indented code, 188-189
indexes, keys, 404
information, namespaces, 386-387
inheriting templates, 332-335
inserting
attributes into templates, 113, 123-124
attribute-sets, 128-129
generated names, 124-127
literal attributes, 127-128
comments into XML documents, 133-135
elements into templates, 113, 115-116
dynamic elements, 119-121
generated names, 118-119
name() function, 121-123
output, 114
tags, 116
XML structure, 116-118
namespaces, 387
nodes with namespaces, 387-389
numbering, 306-310
processing instructions into XML documents, 135-136
text into templates, 108
characters, 108-110
entities, 108-113
unique ID attributes, 415-418
installing
MSXML, 30
MSXSL, 30
Saxon, 31
Xalan-Java, 31
Instant Saxon, 619
instructions, templates, 84
integers, 251
16-bit, 251
32-bit, 251
8-bit, 251
internal DTDs, 413-414
Internet Explorer 5, 38
MSXML, 38
iterating, node-sets, 140
iterations,
variables, creating for, 220-224
XSLT stylesheets, 521

J

Java
extensions
functions, 476-478
functions with scripts, 478-482
xsl-FO to Rtf converter, 173

jfor (Java xsl-FO to Rtf converter), 173

K-L

key() function, 405-412, 419
keys, 404
- advantages, 405
- data, 405-412
- ID attributes, values, 418-419
- indexes, 404
- processing performance, 405
- values, 404

lang attribute, 299-300
- valid values, 300-301

languages, namespaces, 376-377
- advantages, 380
- limitations, 377, 381-383
- namespace names, 378-379
- namespace prefixes, 378-379
- SMIL, 377
- SOAP, 377
- SVG, 377
- WSDL, 377
- XMI, 377
- XUL, 377

last() function, 147
lengths, strings, 277-279
less than sign (&lt), 150
levels, numbering output, 315-316
limitations
- data type conversions, 263
- files, multiple, 320-321
- multiple XML documents, 359-360
- namespace languages, 377
- namespaces, 381
  - DTDs, 382-383
  - languages, mixing, 381
  - Schema, 382-383
  - XML Schemas, 382
- recursion, 427-428
  - memory, 427
  - performance, 427
- stylesheet syntax, 49
- values, comparing, 264
- XSLT, 21
  - computational applications, 22
  - data warehousing applications, 22
  - performance, 21
- XSLT extensions, 467-468

linking
- ID attributes with elements, 415-418
- stylesheets and XML sources, 49-50
  - media, 52
  - multiple stylesheets, 51
  - processing instructions, 50-51
- XML source documents, 355, 357
  - files with elements, 356
  - processing instructions, 356

listings
- (//) expression, 79
- attribute values, 70-71
- Boolean function, false() function, 158
- conditional processing
  - choose-when-otherwise element, 152-153
  - elements without xsl:otherwise, 154
- data, storing in variables, 345-346
- data types
  - conversion, 260
  - variables with tree fragments, 261
- element values, 67-68
- elements, creating from dynamic values, 120-121
- expressions, Boolean operator or, 156-157
- expressions using union operators, 92-93
  - matching nodes, 95
  - values, 94
- functions, sum() function, 447
- HTML output
  - creating, 170
  - sample XML document, 169
  - stylesheet, 169-170

location paths
    ancestor axis, 77-78
    axis, 75-76
namespaces, XML Schema validation, 383
node-set data types, 258
numbering, 307
    composite, 312
    composite on subsets of elements, 313
    parent elements, 311
operators, mod operator, 445
output escaping
    CDATA sections, 177-178
    example, 177
    stylesheet, 177
parameters
    data, adding, 243-244
    HTML data, 233-234
    menus, 241
    stylesheets simplified for reading, 235
    template call, 231
    template functions, 237, 239
    VBScript function, 245
    XML documents, 232
    XML source with menu data, 236
predicates, 69-70
sorting
    descending order, 299
    xsl:apply-templates element, 301
    xsl:for-each element, 297
SQL servers, data, 503
strings
    combining, 270-272
    translate() function, 284
stylesheets
    abc namespace, 379
    adding values together, 271
    attribute-sets, 336-337
    colors, 343-345
    combining data, 348-349
    computational applications, 452-457
    count on multiple elements, 310
    creating documents with namespaces, 388-389
    currency conversion, 461
    duplicate templates, 328
    dynamic sorting, 305
    element-available() function, 493
    embedding, 53
    error handling, 523
    extension elements, 497
    extensions using different processors, 494
    factoral function, 429
    filtering elements on names, 276
    formatted composite numbering, 315-316
    formatting date, 290
    functions, 449
    HTML documents, 392-394
    IDs linking elements, 415-418
    importing, 336
    including, 322
    including with HTML code, 323-327
    internal data, 358
    Java functions, 477-478
    key values, 418
    keys, 406, 408-410
    keys matching elements, 411
    keys, defining, 410
    linked, 50
    local-name() function, 395-396
    matching data in source documents, 349-350
    matching elements, 41-42
    namespace aliases, 390

namespace aliases with Saxon, 391
namespace aliases with Xalan, 391
namespace-uri() function, 387
namespaces, 385-386
numbering, 307-308
numbering pitfalls, 309-310
overriding templates, 332
partial, requiring UTF-16 encoding, 491
pulling data from source documents, 350-352
recursion, 432-437
recursive steps, 430
reversed attribute order, 304
rules for elements, 43
Saxon elements, 471-472
Saxon functions, 473-475
script extension functions, 481
sorting, 297
sorting with templates, 302-303
string functions, 458-459, 461
string lengths, 278
syntax, 48-49
system properties, 490-491
templates, 45
total variable, 450
unique ID, 414
values, selecting, 403-404
whitespace, preserving, 338
whitespace, stripping, 338
writing data from elements, 45-46
writing element ancestors, 47
XML hierarchy, 511-513
XML output, 521
XML with XSLT and VBScript, 54
xsl namespace, 378
xsl:apply-imports, 333-334
xsl:apply-templates element, 44
XSLT versions, 496

substrings, substring-after() function, 282-283
tagged documents, 17
templates
- adding attributes to copied elements, 131
- attribute-sets, 128
- attribute-sets in attribute-sets, 129
- converting attributes to elements, 122
- copying documents, 132
- copying node-sets, 132-133
- default priorities, 102
- dynamic elements, 119-120
- elements, 87-89
- elements, inserting, 114-115
- filter expressions, 145-146
- generated names, 125
- inserting attributes from named templates, 126-127
- inserting elements using xsl.element, 118
- inserting literal attributes, 123-124, 127
- match and named, 97
- matched data, 90
- modes, 98-99
- name() function, 121
- nodes, 85
- output escaping, 109
- priorities, adding, 103
- special characters, 111-112
- XML structure, 116-117

text output method, 171

tree fragments
  in variables, 258
  variables with attributes, 259
values, filtering, 273
variables, 218
  color management, 212-214
  data sets, 220
  nested iterations, 221-222
  sample XML documents, 211
  XSLT elements, 225
whitespace
  deleting from source documents, 180-181
  deleting from values, 184
  example, 179
  output escaping, 187
  partial XML document with JavaScript code, 187
  preserving, 181-183
  stylesheets, 185-186, 188
wildcards, 72-73
  predicates, 74
XML documents, 23
  comments, inserting, 134-135
  computational applications, 451
  data, 400
  file lists, 279
  hierarchy, 510, 512
  internal DTDs, 413
  last() element, 148
  links to Web sites, 274
  mathematical functions, 477
  multiple templates and iteration, 143
  namespaces, mixing, 380, 517
  namespaces, without, 517
  node-sets, filtering, 145
  nodes, filtering with xsl:if, 151
  nodes, processing, 141
  numeric predicates, 147
  processing instructions, inserting, 135-136
  recursion, 432
  strings, 480
  xsl:for-each element, 142-143
  xsl:if element, 148-149
XML documents with one stylesheet, 86
XML output, forcing, 166
XML source with menus, 217
XML sources
  combining data, 347
  currency conversion, 460
  data, retrieving, 342
  file references, 353
  functions, 448
  keys, 406
  linking to files with elements, 355
  linking to files with processing instructions, 356
  multiple namespaces, 394
  namespaces, 384
  recursion, order data, 435
  recursion, totaling, 434
  Saxon elements, 470-471
  sum() function, 447
  values, selecting, 402
  without namespaces, 388
XML tags, 91
XPath expressions, 64
XPath nodes, 65-66
  child nodes, 66-67
xsl:stylesheet element
  empty, 39
  empty output, 39
xsl.copy element, 130
XSLT documents, 24
XSLT tags, 10
XSQL documents
  data queries, 499-500
  stylesheets attached, 501
literal attributes, 127-128
local parameters, 230

local-name() function, 394, 396
localization, format-number() function, 287-288
location paths, 74-75
axis, 75-76
attribute axis, 75
multilevel axis, 77-78, 80
parent axis, 75
node tests, 74

M

Marrowsoft Xselerator, 28
Marrowsoft Xselerator Web site, 28
match attribute, 40-43, 100
matching
data, expressions, 400-401
elements, templates, 40-43
XSLT stylesheets, 521-522
matching nodes, templates, 84
expressions, 92-96
mathematical functions, computational stylesheets, 446-451
mathematical operators, computational stylesheets, 444-446
max function, 239-240
media
output media types, 174
stylesheet processing instructions, 52
media attribute, 52
media-type attribute (output element), 174
memory, recursion, 427
methods, multiple XML documents, 360
Microsoft SQL servers. *See* SQL servers
Microsoft XML. *See* MSXML
mod operators, 444-445
mode attributes, 98
templates, 98-100
Model-View-Controller. *See* MVC
modifying namespaces, 389
aliases, 389-392
MSXML (Microsoft XML), 29-30, 38
installing, 30
versions, 617
MSXML Web site, 38, 618
MSXSL
command-line options, 618
installing, 30
running, 30, 618
MSXSL Web site, 30
msxsl:script element, 480
multifile stylesheets (XSLT), 523
multifiles. *See* multiple, files
multilevel axis, 77-80
multiple
documents (XML), 515-516
files, 320
limitations, 320-321
processors, 486
stylesheets, 320
files, 320-321
variables, creating, 216
XML documents
limitations, 359-360
methods, 360
XML sources, 341
combining data, 347-349
matching data, 349-352
pulling data, 350
multiple options, conditional processing, 152-154
MVC (Model-View-Controller), 518
MVC Web site, 519

N

name attribute, 96
name() function, 121-123
named templates, 96
match templates
combining, 98
compared, 96-98
namespace-uri() function, 386

namespaces, 375-376
advantages, 380-381
languages, mixing, 380
declarations, 489
declaring, 378
information, 386-387
inserting, 387
nodes with namespaces, 387-389
languages, 376-377
limitations, 377
namespace names, 378-379
namespace prefixes, 378-379
SMIL, 377
SOAP, 377
SVG, 377
WSDL, 377
XMI, 377
XUL, 377
limitations, 381
DTDs, 382-383
languages, mixing, 381
Schema, 382-383
XML Schemas, 382
modifying, 389
aliases, 389-392
removing, 387, 392-394
prefixes, 394-396
stylesheets, 37-38
URIs, 380
XML, 14, 516-518
XML sources, processing, 384-386
naming, ID attributes, 414
NaN (Not a Number), 256
negative infinity values, 256
negative numbers, formatted, 287
negative zero values, 255
node sets, deep copy, 131-133
node tests, 74
node-set data types, 257-259
node-set variables, 210
node-sets, 63, 131-133
elements, selecting, 68
filtering, 144, 146
functions, 146-148
iterating, 140
processing, 140-144
nodes, 61
child (XPath), 66-67
context, 70
current, 61-62
inserting with namespaces, 387-389
matching, 84
expressions, 92-96
processing, 140-144
templates, 85
variables, creating, 225-226
XML document trees, 61
current nodes, 61-62
node-sets, 63
nonvalidating parsers, 13
normalize-space function, 183
Not a Number (NaN), 256
not equal sign (!=), 150
not() functions, 158
number data types, 253
conversion to, 261-262
values, 253
ranges, 253
number values, formatted, 288
number() function, 259
numbering, 306
composite, 312-314
grouping, 316
inserting, 306-310
output, 314
multiple levels, 315-316
tokens, 314-316
parent elements, 311
numbers, formatting, 285-286
decimal formatting patterns, 286
default pattern characters, 289
negative numbers, 287
positive numbers, 287
prefixes, 287
special number values, 288
suffixes, 287

O

operations, strings, 270
characters in strings, 272-274
combining strings, 270-272

operators
- (!=) not equal sign, 150
- (&gt) greater than sign, 150
- (&lt) less than sign, 150
- (=) equal sign, 150
- and, 157
- comparison, 150
- computational stylesheets, 444
  - mathematical, 444-446
- div, 444
- mod, 444-445
- or, 157

options
- MSXSL command-line, 618
- Saxon command-line, 619-620
- Xalan command-line, 621-622

or operator, 157

Oracle
- XML data, 498-501
  - client results, 501-502
  - virtual directories, 498
- XSQL Servlet, 498-500

order
- attributes, 488
- sorting, rules, 299-301

order attribute, 298

output, 163-164
- binary data, 171-173
- encoding types
  - determining, 174-175
  - unsupported character sets, 175-176
- escaped characters, 176-179
- HTML output
  - creating, 167-169
  - example, 169-171
  - versions, 169
- media types, 174
- numbering, 314
  - multiple levels, 315-316
  - tokens, 314-316
- parameters, 232-236, 240-242
- PostScript output, 171
- processor extensions, 173-174
- stylesheet imports, 337
- text output method, 171
- text-based formatting, 171
- whitespace, 179
  - deleting, 180-181, 183-185
  - example, 179-180
  - indented code, 188-189
  - preserving, 181-183
  - stylesheets, 185-188
- XML output, 164-165
  - converting to PDF, 173-174
  - external general parsed entities, 164
  - forcing, 165-166
  - versions, 166-167
  - well-formed XML, 164
  - XML declarations, 167
- XSLT stylesheets, 36

output element (XSL), 165-166
- encoding attribute, 175
- example, 166
- media-type attribute, 174

output escaping, disabling, 176-179

output-escaped characters, 113

overriding templates, 332-335

overview (XSLT), 8-9

P

parameters, 230
- advantages, 231-232
- code, 244, 246
- creating, 231
- data, 231
  - adding, 242-244
- data sources, 240
- defined, 230
- duplicate, 329
- global, 230
- importing stylesheets, 339
- local, 230
- output, 232-236, 240-242
- recursion templates, 431-432

template functions, 236-239
templates, 231
values, 231
variables, comparing, 230
XSLT functions, 239-240

parent axis, 75
parent elements, numbering, 311
parsers
nonvalidating, 13
validating, 13
parsers Web site, 29
partial strings, 279-281
PDF format, converting XML to, 173-174
performance
processors, 489
recursion, 427
variables, 209-210
XSLT, 21
position() function, 147, 309
positive infinity values, 255
positive numbers, formatted, 287
positive zero values, 254
PostScript output, 171
predicates, 69
context nodes, 70
expressions, formalizing, 156
wildcards, 74
prefixes
formatted numbers, 287
namespaces, 378-379
removing, 394-396
preserve-space element (XSL), 182-183
preserving whitespace, 181-183
priorities, templates, 100-101
adding, 103-104
default, 101-103
priority attribute, 103
processing
conditional, 148
comparison operators, 150
multiple options, 152-154
xsl:if element, 148-151
node-sets, 140-144
nodes, 140-144
XML sources with namespaces, 384-386
processing instructions
inserting into XML documents, 135-136
linking files, 356
linking stylesheets and XML sources, 50-51
media, 52
multiple stylesheets, 51
XML prologs, 36-37
processor extensions, output formats, 173-174
processors, 22, 29
attribute order, 488
calling, 245-246
character encoding, 488
comparing, 487
elements, 492-493
extensions, 475
functions, 494-495
keys, 405
MSXML, 29-30
multiple, 486
namespace declarations, 489
performance, 489
properties, 490-492
Saxon, 29-31
techniques, 490
Web site, 617
whitespace, 487
Xalan, 29, 31-32
prologs (XML), 36-37
processing instructions, 36-37
properties, processors, 490-492
pull processing, 140
push processing, 140

Q-R

ranges, number data type values, 253
RDDL (Resource Directory Description Language), 380
recursion
defined, 424
functions, 424-425
limitations, 427-428
memory, 427
performance, 427
single values, 432-433
tail, 427

templates, 428
factoral function, 429
parameters, 431-432
recursive steps, 430-431
xsl:call-template, 428, 430
totaling, 434-437
usage, 425-427
relative addressing, 60
removing namespaces, 387, 392-394. ***See also*** **deleting**
prefixes, 394-396
replacing string sections, 283-285
Resource Directory Description Language (RDDL), 380
restructuring (XML), 508
root elements, 14
rules
templates, 40
match attribute, 40-43
XSLT errors, 524-525
running
MSXSL, 30, 618
Saxon, 31, 619
Xalan, 621
Xalan-Java, 32

S

SAX, 15
Saxon, 619
command-line options, 619-620
installing, 31
running, 31, 619
versions, 619
Saxon extension, 468-469
elements, 469-472
functions, 473-475
Saxon processors, 29-31
Saxon Web site, 31, 619
saxon:assign element, 470-472
saxon:expression() function, 474
saxon:sum() element, 473-475
saxon-assign element, 497
SBP (Schema-Based Programming), 518
SBP Web site, 519
Scalable Vector Graphics (SVG), 377
Schema, namespaces, 382-383
Schema-Based Programming (SBP), 518
scripts, extension functions, 478-482
select attribute, 296
selecting
data, expressions, 400-401
values, expressions, 402-404
servers, databases (XML), 497
SGML (Standardized Generalized Markup Language), 12
shallow copy, 129
Simple Object Access Protocol (SOAP), 19, 377
simple variables, 210
single precision, 251
single values, recursion, 432-433
SMIL (Synchronized Multimedia Integration Language), 377
SMIL Web site, 377
SOAP (Simple Object Access Protocol), 19, 377
SOAP Web site, 377
sorting
data, 296
dynamic, 296, 304-306
static, 296
data types, 301
descending order, 298-299
ordering rules, 299-301
xsl:apply-templates, 301-304
xsl:for-each element, 296-298
source documents, 129-133
elements, copying from, 129, 131-133
context node, 129-131
node-sets, 131-133
tree fragments, 131-133
sources (data), parameters, 240

SQL (Structured Query Language), 18
declarative programming languages, 26
SQL servers, XML data, 502-503
Standardized Generalized Markup Language (SGML), 12
starts-with() function, 274-277
static sorting, 296
data types, 301
descending order, 298-299
ordering rules, 299-301
xsl:apply-templates, 301-304
xsl:for-each element, 296-298
storage, data, 16-19, 250
string data type, conversion to, 262
string data types, 256-257
string functions, 458-460
string() function, 259
string-after() function, 283-285
string-before() function, 283-285
string-length() function, 278-279
strings, 269
empty, 279
lengths, 277-279
operations, 270
characters in strings, 272-274
combining strings, 270-272
partial, 279-281
sections, replacing, 283-285
strip-space element (XSL), 180-181
stripping out. *See* deleting
structure
XML, 13
DTDs, 13
inserting elements into templates, 116-118
parsers, 13
XML Schemas, 13
XML document trees, 59
attributes, 60
current nodes, 61-62
elements, 60
node-sets, 63
nodes, 61
Structured Query Language. *See* SQL
stylesheet elements, accessing, 357-359
stylesheets. *See also* xsl:stylesheet; xsl:stylesheet element
computational, 442
functions, 446-451
operators, 444-446
usage, 443
CSS, 36
defined, 36
files with file types, 280
importing, 329-331
attribute-sets, 335-337
decimal formats, 337
output, 337
overriding templates, 332-333, 335
parameters, 339
variables, 339
whitespace, 337, 339
including, 321
comparing with importing, 329-330
duplicate parameters, 329
duplicate templates, 328-329
duplicate variables, 329
elements, 327-328
HTML code, 323-327
loading files in variables, 353-354
multiple, 320
files, 320-321
multiple documents, 86-89, 91
syntax, 48-49
limitations, 49
whitespace, 185-188
XML documents, hierarchy, 513
XML sources, 49
code, 54-55
embedding, 52, 54
linking, 49-52

XSLT, 519-520
attribute-sets, 522
calling, 521
elements, 36
iteration, 521
matching, 521-522
multifile, 523
output, 36
root elements, 37-40
rules, 524-525
troubleshooting errors, 523-524
variables, 522
XML document prolog, 36-37
substring() function, 281
substring-after() function, 281, 283, 394
substring-before() function, 281, 283
substrings, 279-281
characters, 281, 283
after characters, 281, 283
before, 281, 283
suffixes, formatted numbers, 287
sum() function, 447-450
SVG (Scalable Vector Graphics), 377
SVG Web site, 377
Synchronized Multimedia Integration Language. *See* SMIL
syntax
HTML, 48
stylesheets, 48-49
system-property() function, 490-491
xsl:vendor argument, 490
xsl:vendor-url argument, 490
xsl:version argument, 490

T

tableprop attribute-set, 128-129
tags, 10-11
templates, 116
xsl.element, 119
techniques, processors, 490
template functions, parameters, 236-239
templates, 83. *See also* xsl:template
adding, 43-44
advantages, 86
ad hoc data, 91-92
multiple documents with one stylesheet, 86-89, 91
attributes, 113
attribute-sets, 128-129
inserting, 123-124
inserting literal attributes, 127-128
inserting with generated names, 124-127
duplicate, 328-329
elements, 85, 113, 115-116
dynamic elements, 119-121
generated names, 118-119
inserting into output, 114
name() function, 121-123
tags, 116
XML structure, 116-118
firing, 44-45
inheriting, 332-333, 335
instructions, 84
matching nodes, 84
expressions, 92-96
mode attributes, 98-100
named, 96
match, combining, 98
match, compared, 96, 98
nodes, 85
overriding, 332-333, 335
parameters, 231
priorities, 100-101
adding, 103-104
default, 101-103
recursion, 428
factoral function, 429
parameters, 431-432
recursive steps, 430-431
xsl:call-template, 428, 430

rules, 40
match attribute, 40-43
text, 108
characters, 108-110
entities, 108-113
variables, creating for, 220-224

text, templates, 108
characters, 108-110
entities, 108-113

Text Editor, 27

text element (XSL), 186-187

text output method, 171

text() function, 47

tokens, numbering output, 314-316

tools (XML), 518

total variable, 449

totaling, recursion, 434-437

transformation
defined, 9
XML, processors, 22, 29-32
XML documents, 10
XSLT, 23-26

tree fragment data types, 258

tree fragments, 62, 131-133
deep copy, 131-133
variables, 210

trees, XML documents, 59
attributes, 60
current nodes, 61-62
elements, 60
node-sets, 63
nodes, 61

troubleshooting
attribute values, 488
whitespace, 184-185
XSLT errors, 523-524

true() function, 157

types, multilevel axis, 78

U

Uniform Resource Identifier (URI) 14

Uniform Resource Locator (URL), 14

Uniform Resource Name (URN), 14

union operator, 92

unique ID attributes, 414
inserting, 415-418
linking with elements, 415-418

unsupported character sets, encoding, 175-176

URI (Uniform Resource Identifier), 14
Base, 357
namespaces, 380

URL (Uniform Resource Locator), 14

URN (Uniform Resource Name), 14

usage
computational stylesheets, 443
recursion, 425-427
XSLT extensions, 468

use-attribute-sets attribute, 128

UTF-16 encoding, 175

utilities
FOP, 172
Java xsl-FO to Rtf converter, 173

V

validating parsers, 13

values
attributes, 70-72
troubleshooting, 488
comparing, 263-264
conversion rules, 263
disadvantages, 264
elements, 67-70
expressions
formalizing, 155
selecting, 402-404
finite non-zero, 254
keys, 404
ID attributes, 418-419
NaN, 256
negative infinity, 256
negative zero, 255
number data types, 253
ranges, 253
parameters, 231
positive infinity, 255
positive zero, 254
recursion, 432-433
removing whitespace in, 183-184
tree fragments, 62
variables, 210

variables, 208
- accessing, 211
- advantages, 209
  - performance, 209-210
  - values, 210
- colors, 212-214
- complex, 215-216
- creating, 210-211, 213-215
  - data sets, 220-224
  - expressions, 216-220
  - iterations, 220-224
  - multiple, 216
  - nodes, 225-226
  - templates, 220-224
  - XSLT elements, 225
- defined, 208-209
- duplicate, 329
- importing stylesheets, 339
- node-set, 210
- parameters, comparing, 230
- simple, 210
- storing in data, 345-347
- total, 449
- tree fragment, 210
- XSLT, 208, 522
  - changing, 426

version attribute, 36

versions
- HTML output, 169
- MSXML, 617
- Saxon, 619
- Xalan, 620
- XML output, 166-167
- XSLT, elements, 495-497

versions attribute, 37

virtual directories, 498

Visual Studio.NET Web site, 29

Visual XSLT, 29

vocabularies (XML), 13
- namespaces, 14

W-Y

W3 Consortium (W3C), 7

Web Services
- defined, 19
- SOAP, 19
- XML-RPC, 19

Web Services Description Language (WSDL), 377

Web sites
- Apache XML project, 31
- BizTalk, 20
- eXcelon Stylus Studio, 28
- FOP utility, 172
- Java xsl-FO to Rtf converter, 173
- Marrowsoft Xselerator, 28
- MSXML, 38, 618
- MSXSL, 30
- MVC, 519
- parsers, 29
- processors, 617
- Saxon, 31, 619
- SBP, 519
- SMIL, 377
- SOAP, 377
- SVG, 377
- Visual Studio.NET, 29
- Whitehall, 29
- WSDL, 377
- Xalan-Java, 31
- XMI, 377
- XML databases, 19
- XML Namespace Recommendation, 379
- XML Notepad, 27
- XML Pro, 27
- XML Spy, 28
- XPath Visualizer, 155
- XSL debugger, 28
- XUL, 377

well-formed XML, 164

Whitehall <xsl> Composer, 29

Whitehall Web site, 29

whitespace, 179, 337, 339
- deleting
  - from source documents, 180-181
  - from values, 183-184
  - potential problems, 184-185
- example, 179-180
- indented code, 188-189
- preserving, 181-183
- processors, 487
- stylesheets, 185-188

width attribute-set, 128

wildcard characters (*) (XPath), 72-73
- predicates, 74

writing data from elements, 45-47
- text values, 46

WSDL (Web Services Description Language), 377
WSDL Web site, 377

X-Z

Xalan
- command-line options, 621-622
- running, 621
- versions, 620

Xalan Java 2, 620
Xalan processors, 29, 31-32
Xalan-Java
- installing, 31
- running, 32

Xalan-Java Web site, 31
XMI (XML Metadata Interchange), 377
XMI Web site, 377
XML (Extensible Markup Language), 7
- data
  - Oracle, 498-502
  - SQL servers, 502-503
- data storage, 16-19
- database servers, 497
- designing, 508-510
  - attributes, 514-515
  - elements, 514-515
  - hierarchy, 510-514
  - multiple documents, 515-516
  - namespaces, 516-518
  - tools, 518
  - XML Schemas, 518
- restructuring, 508
- SGML, 12
- vocabularies, 13
  - namespaces, 14
- XSLT, 9, 11
  - communicating, 19-21

XML databases Web site, 19
XML declarations, removing from output, 167
XML documents
- comments, inserting, 133-135
- internal DTDs, 413-414
- limitations, 359-360
- methods, 360
- processing instructions, inserting, 135-136
- prologs, 36-37
  - processing instructions, 36-37
- root elements, 37-38
  - namespaces, 37-38
  - xsl:stylesheet element, 39-40
  - xsl:transform element, 38
- transforming, 10
- trees, 59
  - attributes, 60
  - current nodes, 61-62
  - elements, 60
  - node-sets, 63
  - nodes, 61
- whitespace, 179
  - deleting, 180-181, 183-185
  - example, 179-180
  - indented code, 188-189
  - preserving, 181-183
  - stylesheets, 185-188

XML editors, 27-28
XML Metadata Interchange (XMI), 377
XML Namespace Recommendation Web site, 379
XML Notepad, 27
XML Notepad Web site, 27
XML output, 164-165
- converting to PDF, 173-174
- external general parsed entities, 164
- forcing, 165-166
- versions, 166-167
- well-formed XML, 164
- XML declarations, 167

XML Pro, 27
XML Pro Web site, 27
XML Remote Procedure Calling (XML-RPC), 19
XML Schemas, 13
- data types, 252
- namespaces, 382
- XML, 518

XML sources
- accessing, 342
- data, 45, 342-345

combining data, 347-349
matching data, 349-352
pulling data, 350
storing data in variables, 345-347
xsl:value-of element, 45-47
file locations, 357
file references, 353-355
linking documents, 355, 357
files with elements, 356
processing instructions, 356
multiple, 341
namespaces, processing, 384-386
stylesheets, 49
code, 54-55
embedding, 52, 54
linking, 49-52
XML Spy, 28
XML Spy Web site, 28
XML structure, 13
DTDs, 13
inserting elements into templates, 116-118
parsers, 13
XML Schemas, 13
XML transformation, processors, 22, 29
MSXML, 29-30
Saxon, 30-31
Xalan, 31-32
XML-based User Interface Language (XUL), 377
XML-RPC (XML Remote Procedure Calling), 19
xmlns attribute, 378
XPath, 16, 63
attribute values, 70-72
element values, 67-70
expressions, 64-65
child nodes, 66-67
location paths, 74-75
axis, 75-78, 80
node tests, 74
name() function, 121
wildcard characters, 72-73
predicates, 74
XPath Visualizer Web site, 155
XPointer, 16
XSL
XSLFO, 9
XSLT, 9
XSL debugger, 28
XSL debugger Web site, 28
XSL elements
apply-templates, 184
output, 165-166
encoding attribute, 175
example, 166
media-type attribute, 174
preserve-space, 182-183
strip-space, 180-181
text, 186-187
XSL Formatting Objects (XSLFO), 9
XSL Transformations. *See* XSLT
xsl:apply-imports element, 333
xsl:apply-templates element, 43-45, 301-304
xsl:call-template element, 428-430
xsl:choose element, 152
xsl:copy element, 129, 131
xsl:copy-of element, 132-133
xsl:decimal format element, 337
xsl:decimal-format element, 288-289
xsl:fallback element, 496
xsl:for-each element, 142, 296-298
xsl:if element, 148-151
xsl:import element, 330
xsl:include element, 321-322, 488
xsl:key element, 405-409, 411-412
xsl:namespace-alias element, 390-392
xsl:number element, 306-310
xsl:otherwise element, 152
xsl:output element, 337, 488
xsl:preserve-space element, 337, 339
xsl:sort element, 296
order attribute, 298
select attribute, 296
xsl:strip-space element, 337, 339

xsl:stylesheet element, 37, 153, 519
content, 39-40
version attribute, 37
xsl:stylesheets. *See* stylesheets; xsl:stylesheet element
xsl:template. *See* templates
xsl:template element, 40, 153
xsl:transform element, 38
xsl:value-of element, 45-47
xsl:vendor argument, 490
xsl:vendor-url argument, 490
xsl:version argument, 490
xsl:when element, 152
xsl-FO to Rtf converter, 173
xsl.element, 118
xsl.element tag, 119
XSLFO (XSL Formatting Objects), 9
XSLT (Extensible Stylesheet Language Transformation), 7, 9
advantages, 16
data storage, 16-19
data types, 251-252
defined, 9-10
designing, 519
attribute-sets, 522
calling, 521
iteration, 521
matching, 521-522
multifile stylesheets, 523
rules, 524-525
stylesheets, 519-520
troubleshooting errors, 523-524
variables, 522
disadvantages, 21
computational applications, 22
data warehousing applications, 22
performance, 21
DSSSL, 12
extensions, 466
advantages, 466-467
built-in, 468-469
elements, 469-472
extension attributes, 466
extension elements, 466
extension functions, 466
functions, 473-475
limitations, 467-468
usage, 468
overview, 8-9
stylesheets
elements, 36
output, 36
root elements, 37-40
XML document prolog, 36-37
transforming documents, 10
variables, 208
changing, 426
versions, elements, 495-497
XML, 9, 11
communicating, 19-21
XSLT debuggers, 28-29
XSLT design tool, 29
XSLT editors, 28-29
XSLT elements, 225
variables, creating from, 225
nodes, 225-226
XSLT files, 27
Text Editor, 27
XML editors, 27-28
XSLT debuggers, 28-29
XSLT design tools, 29
XSLT editors, 28-29
XSLT functions, parameters, 239-240
XSLT transformation, 23-26
XSQL Servlet, 498-500
XUL (XML-based User Interface Language Language), 377
XUL Web site, 377

Top 5 Common XSLT Mistakes

	PROBLEM	CAUSE
1	The stylesheet doesn't work.	The XSLT namespace is incorrect. It should be `http://www.w3.org/1999/XSL/Transform`. Another common namespace is `http://www.w3.org/TR/WD-xsl`, but this is obsolete, so it shouldn't be used.
2	Although the location path is correct, a selection isn't selecting any data.	The source document is out of context, probably because the current context is a variable with data from outside the source document.
3	A function or expression claims it needs a node-set argument and that the current argument is a tree fragment.	In XSLT 1.0 a tree fragment can't be converted to a node-set. Some processors allow this anyway, others need an extension function.
4	An expression with a number predicate, such as `//dish[6]` doesn't yield a result, although there should be more than six nodes matching.	The precedence of different elements in the expression is wrong, so only a subset is given. The expression should use parentheses to get the right precedence—for instance, `(//dish)[6]`.
5	The stylesheet draws a complete blank.	An extension function or element is misused. This is not properly handled by most processors.

Top 5 XSLT Limitations of XSLT

1 Poor mathematical functionality. Apart from some operators on two values and a function to aggregate values of a node-set, XSLT has no mathematical functions such as calculating the square root, tangent, and so on. Most languages have an abundance of such functions.

2 Poor string functionality. There are several functions to check and manipulate, but nowhere near as much as most other programming languages.

3 No support for date and time. Date and time are key data types for most applications. They are sorely missed in XSLT.

4 No easy way to group data that is related. In data manipulation languages such as SQL, you can select, sort, and group data with a simple expression. That is not possible with XSLT.

5 No ability to create functions in XSLT is callable from XPath. If you could create your own functions in a processor-neutral way, you could get around many of the limitations listed.