

**NEW PERSPECTIVES**

# XML

3<sup>RD</sup> EDITION



COMPREHENSIVE

Carey :: Vodnik

# New Perspectives on XML, 3rd Edition, Comprehensive

## Textbook Reviewers

We are extremely grateful to the *New Perspectives on XML, 3rd Edition, Comprehensive* textbook reviewers listed below, and would like to take this opportunity to acknowledge them for their contributions in the development of this text. Their timely reviews, informed feedback, and excellent suggestions were tremendously valuable and helped us to produce an outstanding text that will meet the needs of all our New Perspectives instructors and students. Our sincere thanks to all!

### Textbook Reviewers

David Doering, St. Louis Community College  
Ravinder Kang, Highline Community College  
Diana Kokoska, University of Maine at Augusta  
Barbara Rader, University of Maryland  
Sheryl Schoenacher, Farmingdale State College  
Dave Sciuto, University of Massachusetts—Lowell  
John Whitney, Fox Valley Technical College  
Dawn Wick, Southwestern Community College

*"The third edition of New Perspectives on XML takes a practical approach to teaching the foundations of XML. It provides real-world scenarios that allow students to work hands-on, applying XML concepts. The structure of each chapter affords the student, whether online or in the classroom, a variety of learning activities designed to support all learning styles. I have successfully used the New Perspectives on XML text for many years, teaching thousands of students the practical purposes for XML. Success in using the text is measurable in the number of 'Aha!' moments most students have thanks to clearly defined and explained concepts, numerous practical examples, and tutorials that challenge their working knowledge of XML. This newest edition takes vital steps toward transforming the student of XML into a working practitioner."*

—Dave Sciuto,  
University of Massachusetts—Lowell

**NEW PERSPECTIVES ON**  
**XML**

*3rd Edition*

---

**COMPREHENSIVE**

**Patrick Carey  
Sasha Vodnik**



---

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit [www.cengage.com/highered](http://www.cengage.com/highered) to search by ISBN#, author, title, or keyword for materials in your areas of interest.

Important Notice: Media content referenced within the product description or the product text may not be available in the eBook version.

**New Perspectives on XML****3rd Edition, Comprehensive**

Product Director: Kathleen McMahon  
Senior Director of Development: Marah Bellegarde  
Senior Product Manager: Jim Gish  
Product Development Manager: Leigh Hefferon  
Senior Content Developer: Kathy Finnegan  
Marketing Director: Michele McTighe  
Senior Marketing Manager: Eric La Scolla  
Developmental Editor: Pam Conrad  
Composition: GEX Publishing Services  
Art Director: Marissa Falco  
Text Designer: Althea Chen  
Cover Designer: GEX Publishing Services  
Cover Art: ©Tarek El Sombati/E+/Getty Images  
Copyeditor: GEX Publishing Services  
Proofreader: Vicki Zimmer  
Indexer: Richard Carlson

© 2015 Cengage Learning

WCN: 02-200-203

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at  
**Cengage Learning Customer & Sales Support, 1-800-354-9706**

For permission to use material from this text or product, submit all requests online at [www.cengage.com/permissions](http://www.cengage.com/permissions)  
Further permissions questions can be emailed to  
[permissionrequest@cengage.com](mailto:permissionrequest@cengage.com)

Library of Congress Control Number: 2014952799

ISBN: 978-1-285-07582-2

**Cengage Learning**  
20 Channel Center Street  
Boston, MA 02210  
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:  
[www.cengage.com/global](http://www.cengage.com/global)

Cengage Learning products are represented in Canada by  
Nelson Education, Ltd.

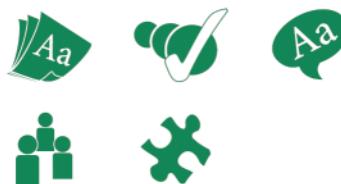
For your course and learning solutions, visit [www.cengage.com](http://www.cengage.com)

Purchase any of our products at your local college store or at our preferred online store [www.cengagebrain.com](http://www.cengagebrain.com)

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Microsoft and the Office logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Cengage Learning is an independent entity from the Microsoft Corporation, and not affiliated with Microsoft in any manner.

Disclaimer: Any fictional data related to persons or companies or URLs used throughout this book is intended for instructional purposes only. At the time this book was printed, any such data was fictional and not belonging to any real persons or companies.



ProSkills Icons © 2014 Cengage Learning.

Printed in the United States of America

Print Number: 01

Print Year: 2014

# Preface

The New Perspectives Series' critical-thinking, problem-solving approach is the ideal way to prepare students to transcend point-and-click skills and take advantage of all that XML has to offer.

In developing the New Perspectives Series, our goal was to create books that give students the software concepts and practical skills they need to succeed beyond the classroom. We've updated our proven case-based pedagogy with more practical content to make learning skills more meaningful to students. With the New Perspectives Series, students understand *why* they are learning *what* they are learning, and are fully prepared to apply their skills to real-life situations.

*"With the clear instructions in this text, students know exactly what code to write and where to place it. The exercises enable students to apply what they've learned using realistic business scenarios. I would recommend this text to anyone teaching XML or learning it on their own."*

—Dawn Wick  
Southwestern Community  
College

## About This Book

This book provides complete coverage of XML including the following:

- Using XSLT to transform XML data into HTML format
- Creating custom reports using XSLT 2.0 and XPath 2.0
- Designing database queries using XQuery

*New for this edition!*

- Each session begins with a Visual Overview, which includes colorful, enlarged figures with numerous callouts and key term definitions, giving students a comprehensive preview of the topics covered in the session, as well as a handy study guide.
- New ProSkills boxes provide guidance for how to use the software in real-world, professional situations, and related ProSkills exercises integrate the technology skills students learn with one or more of the following soft skills: decision making, problem solving, teamwork, verbal communication, and written communication.
- Important steps are highlighted in yellow with attached margin notes to help students pay close attention to completing the steps correctly and avoid time-consuming rework.

## System Requirements

This book assumes that students have access to a current browser that supports the viewing of XML files and XML files transformed using XSLT. Current versions of the major browsers support these features of XML with the exception of Google Chrome, which does not support XML documents stored locally. The screenshots of web pages in this book were produced using Internet Explorer 10 running on Windows 7 Professional (64-bit) and Internet Explorer 11 running on Windows 8.1 (64-bit), unless otherwise noted. Students who intend to validate their XML documents in Tutorials 2 through 4 should have access to an XML validating parser, such as Exchanger XML Editor, or to an online validation service. Students who intend to transform XML documents using XSLT should have access to an XSLT processor such as Exchanger, XMLSpy or Saxon. The transformations performed in Tutorials 5 through 8 were done using Saxon-HE (home edition) available free for Java or .NET at <http://saxon.sourceforge.net>. Students who perform XQuery data queries in Tutorial 9 should have access to an XQuery processor. Such queries were performed in Tutorial 9 using Saxon-HE. Students who are using processors other than Saxon should consult their processor's documentation for specific installation and operation instructions.

[www.cengage.com/series/newperspectives](http://www.cengage.com/series/newperspectives)

"New Perspectives texts provide up-to-date, real-world application of content, making book selection easy. The step-by-step, hands-on approach teaches students concepts they can apply immediately."

—John Taylor  
Southeastern Technical College

## VISUAL OVERVIEW

# The New Perspectives Approach

### Context

Each tutorial begins with a problem presented in a "real-world" case that is meaningful to students. The case sets the scene to help students understand what they will do in the tutorial.

### Hands-on Approach

Each tutorial is divided into manageable sessions that combine reading and hands-on, step-by-step work. Colorful screenshots help guide students through the steps. **Trouble?** tips anticipate common mistakes or problems to help students stay on track and continue with the tutorial.

### Visual Overviews

*New for this edition!* Each session begins with a Visual Overview, a new two-page spread that includes colorful, enlarged figures with numerous callouts and key term definitions, giving students a comprehensive preview of the topics covered in the session, as well as a handy study guide.

## PROSKILLS

### ProSkills Boxes and Exercises

*New for this edition!* ProSkills boxes provide guidance for how to use the software in real-world, professional situations, and related ProSkills exercises integrate the technology skills students learn with one or more of the following soft skills: decision making, problem solving, teamwork, verbal communication, and written communication.

## KEY STEP

### Key Steps

*New for this edition!* Important steps are highlighted in yellow with attached margin notes to help students pay close attention to completing the steps correctly and avoid time-consuming rework.

## INSIGHT

### InSight Boxes

InSight boxes offer expert advice and best practices to help students achieve a deeper understanding of the concepts behind the software features and skills.

## TIP

### Margin Tips

Margin Tips provide helpful hints and shortcuts for more efficient use of the software. The Tips appear in the margin at key points throughout each tutorial, giving students extra information when and where they need it.

## REVIEW

## APPLY

### Assessment

Retention is a key component to learning. At the end of each session, a series of Quick Check questions helps students test their understanding of the material before moving on. Engaging end-of-tutorial Review Assignments and Case Problems have always been a hallmark feature of the New Perspectives Series. Colorful bars and brief descriptions accompany the exercises, making it easy to understand both the goal and level of challenge a particular assignment holds.

## REFERENCE

## GLOSSARY/INDEX

### Reference

Within each tutorial, Reference boxes appear before a set of steps to provide a succinct summary and preview of how to perform a task. In addition, each book includes a combination Glossary/Index to promote easy reference of material.

## Our Complete System of Instruction

BRIEF

INTRODUCTORY

COMPREHENSIVE

COURSECASTS

### Coverage To Meet Your Needs

Whether you're looking for just a small amount of coverage or enough to fill a semester-long class, we can provide you with a textbook that meets your needs.

- Brief books typically cover the essential skills in just 2 to 4 tutorials.
- Introductory books build and expand on those skills and contain an average of 5 to 8 tutorials.
- Comprehensive books are great for a full-semester class, and contain 9 to 12+ tutorials.

So if the book you're holding does not provide the right amount of coverage for you, there's probably another offering available. Go to our Web site or contact your Cengage Learning sales representative to find out what else we offer.

### CourseCasts – Learning on the Go. Always available...always relevant.

Want to keep up with the latest technology trends relevant to you? Visit <http://coursecasts.course.com> to find a library of weekly updated podcasts, CourseCasts, and download them to your mp3 player.

Ken Baldauf, host of CourseCasts, is a faculty member of the Florida State University Computer Science Department where he is responsible for teaching technology classes to thousands of FSU students each year. Ken is an expert in the latest technology trends; he gathers and sorts through the most pertinent news and information for CourseCasts so your students can spend their time enjoying technology, rather than trying to figure it out. Open or close your lecture with a discussion based on the latest CourseCast.

Visit us at <http://coursecasts.course.com> to learn on the go!

### Instructor Resources

We offer more than just a book. We have all the tools you need to enhance your lectures, check students' work, and generate exams in a new, easier-to-use and completely revised package. This book's Instructor's Manual, Cengage Learning Testing Powered by Cognero, PowerPoint presentations, data files, solution files, figure files, and a sample syllabus are all available on this text's Instructor Companion Site. Simply search for this text at [login.cengage.com](http://login.cengage.com).



### SAM: Skills Assessment Manager

Get your students workplace-ready with SAM, the premier proficiency-based assessment and training solution for Microsoft Office! SAM's active, hands-on environment helps students master computer skills and concepts that are essential to academic and career success.

Skill-based assessments, interactive trainings, business-centric projects, and comprehensive remediation engage students in mastering the latest Microsoft Office programs on their own, allowing instructors to spend class time teaching. SAM's efficient course setup and robust grading features provide faculty with consistency across sections. Fully interactive MindTap Readers integrate market-leading Cengage Learning content with SAM, creating a comprehensive online student learning environment.

[www.cengage.com/series/newperspectives](http://www.cengage.com/series/newperspectives)

## Acknowledgments

I would like to thank the people who worked so hard to make this book possible. Special thanks to my developmental editor, Pam Conrad, for her excellent hard work and dedication in editing this text, and to my Content Developer, Kathy Finnegan, who has worked tirelessly in overseeing this project and made my task so much easier with her enthusiasm and good humor. Other people at Cengage Learning who deserve credit are Jim Gish, Senior Product Manager; Christian Kunciw, Manuscript Quality Assurance (MQA) Supervisor; and John Freitas, Serge Palladino, Danielle Shaw, and Susan Whalen, MQA testers.

Feedback is an important part of writing any book, and thanks go to the following reviewers for their helpful ideas and comments: David Doering, St. Louis Community College; Ravinder Kang, Highline Community College; Diana Kokoska, University of Maine at Augusta; Barbara Rader, University of Maryland; Sheryl Schoenacher, Farmingdale State College; Dave Sciuto, University of Massachusetts—Lowell; John Whitney, Fox Valley Technical College; and Dawn Wick, Southwestern Community College.

I want to thank my wife Joan for her support during this project and for my six children to whom this book is dedicated.

— Patrick Carey

Many thanks to everyone who helped in this revision. Pam Conrad, my sharp-eyed developmental editor, suggested improvements and asked a lot of important questions that helped me immeasurably in tightening up the material. The good advice of Kathy Finnegan, my Content Developer, kept me focused on the important aspects of the revision process, and she sweated a lot of the small stuff so I didn't have to. I'm also grateful to Jim Gish, the Senior Product Manager, for keeping the faith during the evolution of this revision. The staff at GEX Publishing Services made it all look amazing. And MQA testers Serge Palladino, Danielle Shaw, and Susan Whalen read everything through, completed all the steps, and gave smart feedback that removed many roadblocks for future users. Finally, thanks to my husband, Jason Bucy, for encouraging me to balance diving deep into XML with stepping away from the computer, getting outside, and enjoying the world with him.

— Sasha Vodnik

# BRIEF CONTENTS

## XML

<b>Tutorial 1</b> Creating an XML Document.....	XML 1
<i>Developing a Document for SJB Pet Boutique</i>	
<b>Tutorial 2</b> Validating Documents with DTDs .....	XML 65
<i>Creating a Document Type Definition for Map Finds For You</i>	
<b>Tutorial 3</b> Validating Documents with Schemas.....	XML 129
<i>Creating a Schema for the ATC School of Information Technology</i>	
<b>Tutorial 4</b> Working with Advanced Schemas .....	XML 195
<i>Creating Advanced Schemas for Higher Ed Test Prep</i>	
<b>Tutorial 5</b> Transforming XML with XSLT and XPath .....	XML 251
<i>Writing XML Data to an Output File</i>	
<b>Tutorial 6</b> Functional Programming with XSLT and XPath 1.0 .....	XML 323
<i>Designing a Product Review Page</i>	
<b>Tutorial 7</b> Building an XSLT Application.....	XML 399
<i>Working with IDs, Keys, and Groups</i>	
<b>Tutorial 8</b> Building Applications with XSLT 2.0.....	XML 457
<i>Exploring XSLT 2.0 and XPath 2.0</i>	
<b>Tutorial 9</b> Exploring Data with XQuery .....	XML 529
<i>Querying Sales Totals from a Database</i>	
<b>Appendix A</b> XML Schema Reference.....	XML A1
<b>Appendix B</b> DTD Reference.....	XML B1
<b>Appendix C</b> XSLT Elements and Attributes .....	XML C1
<b>Appendix D</b> XPath Reference .....	XML D1
<b>Appendix E</b> Using Saxon for XSLT and XQuery.....	XML E1
<b>Appendix F</b> Understanding Regular Expressions.....	XML F1
<b>Glossary/Index</b>	REF 1

# TABLE OF CONTENTS

Preface .....	iii	SESSION 1.2.....	XML 22
<b>Tutorial 1 Creating an XML Document</b>			
<i>Developing a Document for SJB Pet     Boutique</i> .....	<b>XML 1</b>	Working with Elements .....	XML 24
<b>SESSION 1.1.....</b> <b>XML 2</b>			
Introducing XML .....	XML 4	Empty Elements .....	XML 25
The Roots of XML.....	XML 4	Nesting Elements .....	XML 25
XML Today .....	XML 4	The Element Hierarchy .....	XML 26
XML with Software Applications and Languages.....	XML 5	Charting the Element Hierarchy .....	XML 28
XML and Databases .....	XML 5	Writing the Document Body.....	XML 30
XML and Mobile Development.....	XML 6	Working with Attributes .....	XML 32
Creating an XML Vocabulary .....	XML 7	Using Character and Entity References.....	XML 35
Standard XML Vocabularies .....	XML 8	Understanding Text Characters and White Space.....	XML 39
DTDs and Schemas.....	XML 10	Parsed Character Data .....	XML 39
Well-Formed and Valid XML Documents .....	XML 10	Character Data .....	XML 39
Creating an XML Document .....	XML 11	White Space .....	XML 40
The Structure of an XML Document .....	XML 11	Creating a CDATA Section .....	XML 40
The XML Declaration .....	XML 12	Formatting XML Data with CSS .....	XML 44
Inserting Comments .....	XML 14	Applying a Style to an Element .....	XML 45
Processing an XML Document .....	XML 16	Inserting a Processing Instruction.....	XML 46
XML Parsers .....	XML 16	Working with Namespaces .....	XML 49
Session 1.1 Quick Check .....	XML 21	Declaring a Namespace .....	XML 49
Applying a Default Namespace .....	XML 49	Review Assignments .....	XML 52
Case Problems.....	XML 54	Session 1.2 Quick Check .....	XML 51

<b>Tutorial 2 Validating Documents with DTDs</b>	<b>SESSION 2.3.....</b>	<b>XML 104</b>
<i>Creating a Document Type Definition for Map Finds For You .....</i>	Introducing Entities .....	XML 106
<b>SESSION 2.1.....</b>	Working with General Entities.....	XML 106
Creating a Valid Document .....	Creating Parsed Entities .....	XML 107
Declaring a DTD .....	Referencing a General Entity .....	XML 108
Writing the Document Type Declaration .....	Working with Parameter Entities.....	XML 113
Declaring Document Elements .....	Inserting Comments into a DTD .....	XML 115
Elements Containing Any Type of Content .....	Creating Conditional Sections.....	XML 116
Empty Elements .....	Working with Unparsed Data .....	XML 117
Elements Containing Parsed Character Data.....	Validating Standard Vocabularies .....	XML 119
Working with Child Elements .....	Session 2.3 Quick Check .....	XML 121
Specifying an Element Sequence .....	Review Assignments .....	XML 122
Specifying an Element Choice .....	Case Problems.....	XML 123
Modifying Symbols .....	<b>Tutorial 3 Validating Documents with Schemas</b>	
Session 2.1 Quick Check .....	<i>Creating a Schema for the ATC School of Information Technology .....</i>	<b>XML 129</b>
<b>SESSION 2.2.....</b>	<b>SESSION 3.1.....</b>	<b>XML 130</b>
Declaring Attributes .....	Introducing XML Schema .....	XML 132
Working with Attribute Types .....	The Limits of DTDs .....	XML 133
Character Data .....	Schemas and DTDs.....	XML 133
Enumerated Types .....	Schema Vocabularies .....	XML 134
Tokenized Types .....	Starting a Schema File.....	XML 135
Working with Attribute Defaults .....	Understanding Simple and Complex Types .....	XML 137
Validating an XML Document .....	Defining a Simple Type Element.....	XML 138
Session 2.2 Quick Check .....	Defining an Attribute.....	XML 139

Defining a Complex Type Element .....	XML 141
Defining an Element Containing Only Attributes .....	XML 142
Defining an Element Containing Attributes and Basic Text .....	XML 142
Referencing an Element or Attribute Definition.....	XML 143
Defining an Element with Nested Children.....	XML 145
Defining an Element Containing Nested Elements and Attributes.....	XML 147
Indicating Required Attributes .....	XML 150
Specifying the Number of Child Elements .....	XML 152
Validating a Schema Document .....	XML 153
Applying a Schema to an Instance Document .....	XML 155
Session 3.1 Quick Check .....	XML 159
<b>SESSION 3.2.....</b>	<b>XML 160</b>
Validating with Built-In Data Types .....	XML 162
String Data Types .....	XML 163
Numeric Data Types .....	XML 164
Data Types for Dates and Times.....	XML 165
Deriving Customized Data Types .....	XML 168
Deriving a List Data Type .....	XML 170
Deriving a Union Data Type .....	XML 170
Deriving a Restricted Data Type.....	XML 171
Deriving Data Types Using Regular Expressions .....	XML 177
Introducing Regular Expressions .....	XML 178
Applying a Regular Expression.....	XML 180
Session 3.2 Quick Check.....	XML 183
Review Assignments .....	XML 184
Case Problems.....	XML 185
<b>Tutorial 4 Working with Advanced Schemas</b>	
<i>Creating Advanced Schemas for Higher Ed</i>	
Test Prep .....	<b>XML 195</b>
<b>SESSION 4.1.....</b>	<b>XML 196</b>
Designing a Schema .....	XML 198
Flat Catalog Design .....	XML 198
Russian Doll Design.....	XML 200
Venetian Blind Design.....	XML 202
Session 4.1 Quick Check .....	XML 205
<b>SESSION 4.2.....</b>	<b>XML 206</b>
Combining XML Vocabularies.....	XML 208
Creating a Compound Document .....	XML 210
Understanding Name Collision.....	XML 212
Working with Namespaces in an Instance Document .....	XML 213
Declaring and Applying a Namespace to a Document.....	XML 213
Applying a Namespace to an Element.....	XML 215
Working with Attributes .....	XML 217

Associating a Schema with a Namespace . . . . .	XML 219	Introducing XSLT Templates . . . . .	XML 263
Targeting a Namespace . . . . .	XML 219	The Root Template . . . . .	XML 263
Including and Importing Schemas . . . . .	XML 222	Literal Result Elements . . . . .	XML 264
Referencing Objects from Other Schemas . . . . .	XML 223	Defining the Output Format . . . . .	XML 266
Combining Standard Vocabularies . . . . .	XML 225	Transforming a Document . . . . .	XML 268
Session 4.2 Quick Check . . . . .	XML 227	Running Transformations Using Saxon . . . . .	XML 269
<b>SESSION 4.3 . . . . .</b>	<b>XML 228</b>	Session 5.1 Quick Check . . . . .	XML 273
Adding a Namespace to a Style Sheet . . . . .	XML 230	<b>SESSION 5.2 . . . . .</b>	<b>XML 274</b>
Declaring a Namespace in a Style Sheet . . . . .	XML 232	Extracting Element Values . . . . .	XML 276
Qualifying Elements and Attributes by Default . . . . .	XML 235	Using the <code>for-each</code> Element . . . . .	XML 280
Session 4.3 Quick Check . . . . .	XML 239	Working with Templates . . . . .	XML 281
Review Assignments . . . . .	XML 240	Applying a Template . . . . .	XML 282
Case Problems . . . . .	XML 241	Displaying Attribute Values . . . . .	XML 285
ProSkills Exercise: Decision Making . . . . .	XML 248	Combining Node Sets . . . . .	XML 288
<b>Tutorial 5 Transforming XML with XSLT and XPath</b>		Session 5.2 Quick Check . . . . .	XML 291
Writing XML Data to an Output File . . . . .	<b>XML 251</b>	<b>SESSION 5.3 . . . . .</b>	<b>XML 292</b>
<b>SESSION 5.1 . . . . .</b>	<b>XML 252</b>	Inserting a Value into an Attribute . . . . .	XML 294
Introducing XSL and XSLT . . . . .	XML 254	Sorting Node Sets . . . . .	XML 295
XSLT Style Sheets and Processors . . . . .	XML 254	Conditional Processing . . . . .	XML 297
Attaching an XSLT Style Sheet . . . . .	XML 255	Using Comparison Operators and Functions . . . . .	XML 298
Starting an XSLT Style Sheet . . . . .	XML 258	Testing for Multiple Conditions . . . . .	XML 299
Introducing XPath . . . . .	XML 259	Filtering XML with Predicates . . . . .	XML 302
Working with Nodes . . . . .	XML 259	Predicates and Node Position . . . . .	XML 303
Absolute and Relative Location Paths . . . . .	XML 260	Predicates and Functions . . . . .	XML 303
Text, Comment, and Process Instruction Nodes . . . . .	XML 262		

Constructing Elements and Attributes with XSLT .....	XML 307	Accessing an External Style Sheet .....	XML 339
Constructing an Element Node .....	XML 308	Including a Style Sheet .....	XML 339
Constructing Attributes and Attribute Sets .....	XML 309	Importing a Style Sheet .....	XML 339
Constructing Comments and Processing Instructions .....	XML 310	Session 6.1 Quick Check .....	XML 341
Session 5.3 Quick Check .....	XML 312	<b>SESSION 6.2 .....</b>	<b>XML 342</b>
Review Assignments .....	XML 313	Creating a Lookup Table in XSLT .....	XML 344
Case Problems .....	XML 315	Working with Numeric Functions .....	XML 347
<b>Tutorial 6 Functional Programming with XSLT and XPath 1.0</b>		Applying Mathematical Operators .....	XML 349
<i>Designing a Product Review Page .....</i>	<b>XML 323</b>	Numerical Calculations in XPath 2.0 .....	XML 351
<b>SESSION 6.1 .....</b>	<b>XML 324</b>	Formatting Numeric Values .....	XML 351
Using XSLT Variables .....	XML 326	Using the <code>format-number()</code> Function ..	XML 352
Creating a Variable .....	XML 326	International Number Formats .....	XML 352
Understanding Variable Scope .....	XML 327	Working with Text Strings .....	XML 355
Applying a Variable .....	XML 327	Extracting and Combining Text Strings .....	XML 355
Referencing a Variable .....	XML 329	Formatting a Date String .....	XML 356
Copying Nodes .....	XML 331	Working with White Space .....	XML 360
The <code>copy</code> Element .....	XML 332	Session 6.2 Quick Check .....	XML 361
The <code>copy-of</code> Element .....	XML 333	<b>SESSION 6.3 .....</b>	<b>XML 362</b>
Retrieving Data from Multiple Files .....	XML 335	Introducing Parameters .....	XML 364
The <code>document()</code> and <code>doc()</code> Functions ..	XML 335	Setting a Global Parameter Value .....	XML 365
Applying the <code>document()</code> Function .....	XML 336	Exploring Template Parameters .....	XML 367
Retrieving Data from a non-XML File .....	XML 338	Using Named Templates .....	XML 368
		Introducing Functional Programming .....	XML 369
		Understanding Recursion .....	XML 371
		Creating a Recursive Template .....	XML 372
		Session 6.3 Quick Check .....	XML 384

Review Assignments .....	XML 385	Testing Function Availability.....	XML 443		
Case Problems.....	XML 387	Testing Element Availability .....	XML 444		
<b>Tutorial 7 Building an XSLT Application</b>					
Working with IDs, Keys, and Groups.....	<b>XML 399</b>	Session 7.2 Quick Check .....	XML 445		
<b>SESSION 7.1 .....</b> <b>XML 400</b>					
Working with Step Patterns.....	XML 402	Review Assignments .....	XML 446		
Working with Axes .....	XML 403	Case Problems.....	XML 449		
Creating Unique Lists with Step Patterns..	XML 406	<b>Tutorial 8 Building Applications with XSLT 2.0</b>			
Using the mode Attribute.....	XML 411	Exploring XSLT 2.0 and XPath 2.0.....	<b>XML 457</b>		
Session 7.1 Quick Check .....	XML 415	<b>SESSION 8.1 .....</b> <b>XML 458</b>			
<b>SESSION 7.2 .....</b> <b>XML 416</b>					
Working with IDs.....	XML 418	Introducing XSLT 2.0 .....	XML 460		
Generating an ID Value.....	XML 419	Overview of XSLT 2.0 .....	XML 460		
Working with Keys.....	XML 420	Creating an XSLT 2.0 Style Sheet .....	XML 460		
Creating a Key.....	XML 420	Transforming Data from Multiple Sources.....	XML 461		
Applying the key( ) Function .....	XML 422	Using a Catalog File .....	XML 462		
Using Keys with External Documents .....	XML 424	Applying the collection( ) Function .....	XML 462		
Organizing Nodes with Muenchian Grouping... Creating Links with Generated IDs .....	XML 430	Exploring Atomic Values and Data Types .....	XML 467		
Introducing Extension Functions.....	XML 439	Constructor Functions .....	XML 467		
Defining the Extension Namespace.....	XML 440	Displaying Dates and Times .....	XML 468		
Using an Extension Function .....	XML 440	Formatting Date Values .....	XML 471		
Writing an Extension Function .....	XML 441	Calculating Date and Time Durations.....	XML 475		
Applying Extension Elements and Attributes... Changing a Variable's Value .....	XML 442	Introducing Sequences .....	XML 478		
Creating a Loop .....	XML 443	Sequences and Node Sets .....	XML 479		
		Looping Through a Sequence.....	XML 480		
		Sequence Operations.....	XML 481		
		Session 8.1 Quick Check .....	XML 483		

<b>SESSION 8.2 . . . . .</b>	<b>XML 484</b>		
Creating Functions with XSLT 2.0 . . . . .	XML 486	Writing a Path Expression . . . . . XML 538	
Writing an XSLT 2.0 Function . . . . .	XML 487	Running a Query . . . . . XML 539	
Running an XSLT 2.0 Function . . . . .	XML 488	Formatting the Query Output . . . . . XML 539	
Creating Conditional Expressions in XPath 2.0 . .	XML 489	Adding Elements and Attributes to a Query Result. . . . . XML 541	
Grouping Data in XSLT 2.0 . . . . .	XML 492	Declaring XQuery Variables. . . . . XML 543	
Grouping Methods . . . . .	XML 492	Using External Variables . . . . . XML 545	
Applying Styles to a Group. . . . .	XML 494	Introducing FLWOR . . . . . XML 548	
Creating Lookup Tables with the doc() Function . . . . .	XML 497	The Syntax of FLWOR . . . . . XML 548	
Session 8.2 Quick Check . . . . .	XML 499	Working with the <code>for</code> Clause . . . . . XML 549	
<b>SESSION 8.3 . . . . .</b>	<b>XML 500</b>	Defining Variables with the <code>let</code> Clause . . . . . XML 550	
Working with Text Strings in XSLT 2.0 . . . . .	XML 502	Filtering with the <code>where</code> Clause . . . . . XML 551	
Regular Expressions with XPath 2.0 . . . . .	XML 503	Sorting with the <code>order by</code> Clause . . . . . XML 551	
Analyzing Text Strings in XSLT 2.0 . . . . .	XML 505	Displaying Results with the <code>return</code> Clause . . . . . XML 552	
Importing non-XML Data . . . . .	XML 507	Writing a FLWOR Query . . . . . XML 552	
Reading from an Unparsed Text File . . . . .	XML 508	Calculating Summary Statistics . . . . . XML 556	
Session 8.3 Quick Check . . . . .	XML 516	Session 9.1 Quick Check . . . . . XML 559	
Review Assignments . . . . .	XML 517	<b>SESSION 9.2 . . . . .</b>	<b>XML 560</b>
Case Problems. . . . .	XML 520	Joining Data from Multiple Files . . . . . XML 562	
<b>Tutorial 9 Exploring Data with XQuery</b>		Querying Two Source Documents . . . . . XML 562	
<i>Querying Sales Totals from a Database . . . . .</i>	<b>XML 529</b>	Querying Three Source Documents . . . . . XML 563	
<b>SESSION 9.1 . . . . .</b>	<b>XML 530</b>	Grouping the Query Results . . . . . XML 565	
Introducing XQuery . . . . .	XML 532	Grouping by Distinct Values . . . . . XML 568	
Writing an XQuery Document . . . . .	XML 532	Summary Statistics by Group . . . . . XML 568	
Declarations in the Query Prolog . . . . .	XML 532	Declaring a Function . . . . . XML 570	
Commenting Text in XQuery . . . . .	XML 534	Calling a User-Defined Function . . . . . XML 573	

Creating a Query Library Module .....	XML 574
Exploring Library Modules .....	XML 575
Importing a Module .....	XML 577
Moving to XQuery 3.0 .....	XML 580
Grouping Query Results .....	XML 580
The count Clause .....	XML 580
Try and Catch Errors .....	XML 581
Session 9.2 Quick Check .....	XML 583
Review Assignments .....	XML 584
Case Problems.....	XML 586
ProSkills Exercise: Problem Solving.....	XML 594

**Appendix A XML Schema Reference. .... XML A1**

XML Schema Built-In Data Types.....	XML A1
XML Schema Elements .....	XML A3
XML Schema Facets .....	XML A9

**Appendix B DTD Reference ..... XML B1**

Element Declarations .....	XML B2
Attribute Declarations.....	XML B2
Notation Declarations.....	XML B3
Parameter Entity Declarations .....	XML B3
General Entity Declarations .....	XML B4

**Appendix C XSLT Elements and Attributes... XML C1**

Appendix D XPath Reference .....	XML D1
Location Paths .....	XML D1

XPath Operators .....	XML D3
XPath Parameters and Functions .....	XML D4

**Appendix E Using Saxon for XSLT and XQuery..... XML E1**

Getting Started with Saxon.....	XML E2
Installing Java .....	XML E2
Installing Saxon.....	XML E3
Setting the CLASSPATH Variable .....	XML E3
Running Transformations from the Command Line .....	XML E5
Running Queries from the Command Line .....	XML E8

**Appendix F Understanding Regular Expressions ..... XML F1**

Writing a Regular Expression .....	XML F2
Matching a Substring .....	XML F2
Regular Expression Modifiers.....	XML F2
Defining Character Positions .....	XML F2
Defining Character Types and Character Classes .....	XML F4
Using Character Classes .....	XML F5
Specifying Character Repetition.....	XML F6
Using Escape Sequences .....	XML F7
Specifying Alternate Patterns and Grouping	XML F8

**GLOSSARY/INDEX..... REF 1**

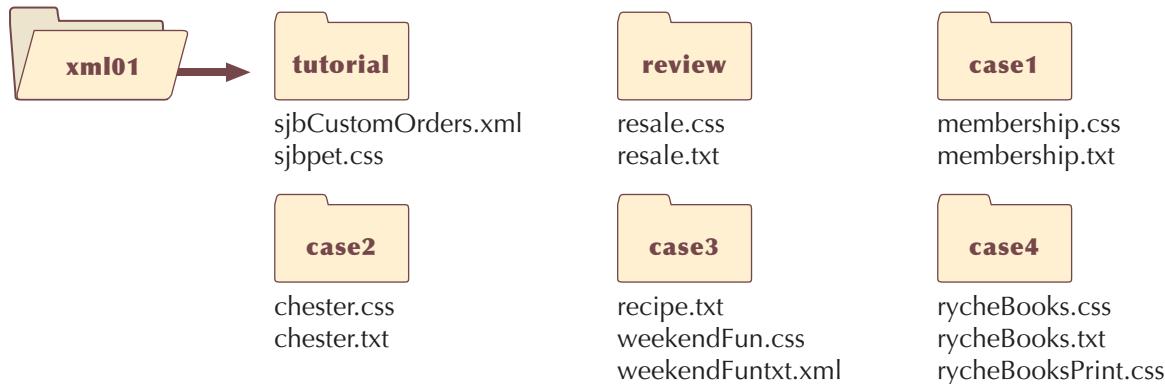


**OBJECTIVES****Session 1.1**

- Describe the history of XML and the uses of XML documents
- Understand XML vocabularies
- Define well-formed and valid XML documents, and describe the basic structure of an XML document
- Create an XML declaration
- Work with XML comments
- Work with XML parsers and understand how web browsers work with XML documents

**Session 1.2**

- Create XML elements and attributes
- Work with character and entity references
- Describe how XML handles parsed character data, character data, and white space
- Create an XML processing instruction to apply a style sheet to an XML document
- Declare a default namespace for an XML vocabulary and apply the namespace to an element

**STARTING DATA FILES**

# Creating an XML Document

## *Developing a Document for SJB Pet Boutique*

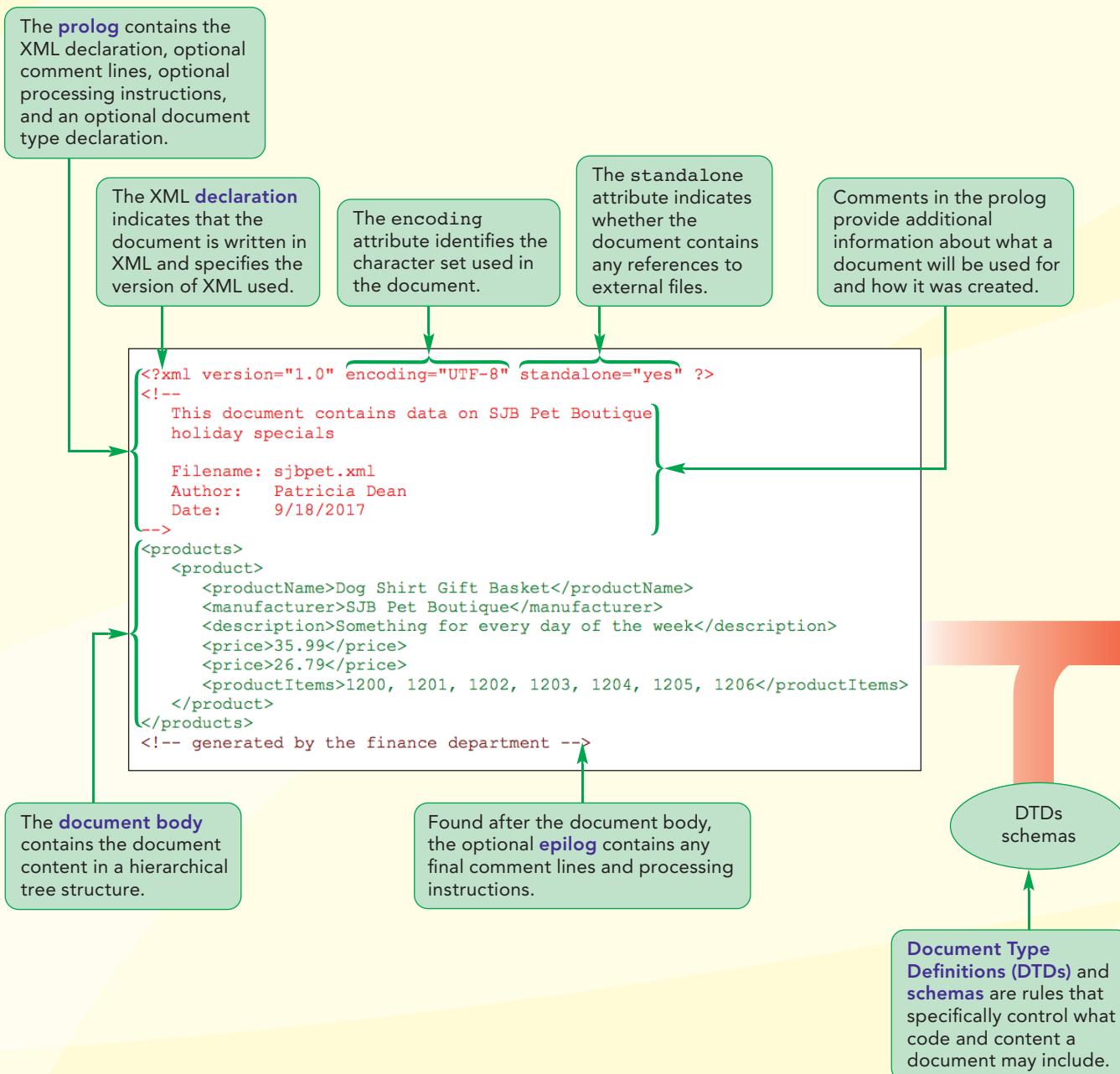
### **Case | SJB Pet Boutique**

SJB Pet Boutique in Delafield, Wisconsin, creates beautiful jewelry and clothing accessories “for pets and their humans.” The boutique’s top two best-selling items are holiday pet costumes, and matching pet collar and human necklace pendants.

During the past year, the boutique has received more requests for custom work. The owners would like to further develop this aspect of their business by making it available on the SJB Pet Boutique website. Patricia Dean manages the boutique’s website. She has been investigating using Extensible Markup Language to organize information about the boutique’s product line and the custom work offered. **Extensible Markup Language (XML)** is a markup language that can be extended and modified to match the needs of the document author and the data being recorded. XML has some advantages in presenting structured content such as descriptions of available customizations. Data stored in an XML document can be integrated with the boutique’s website. Through the use of style sheets, Patricia can present XML data in a way that would be attractive to potential customers.

The boutique’s website already takes advantage of many of the latest web standards, including HTML5 and CSS. Patricia would like to gradually incorporate XML into the website and increase the use of style sheets. As a first step, she has asked for your help in creating a document that will display a small part of the boutique’s inventory using XML.

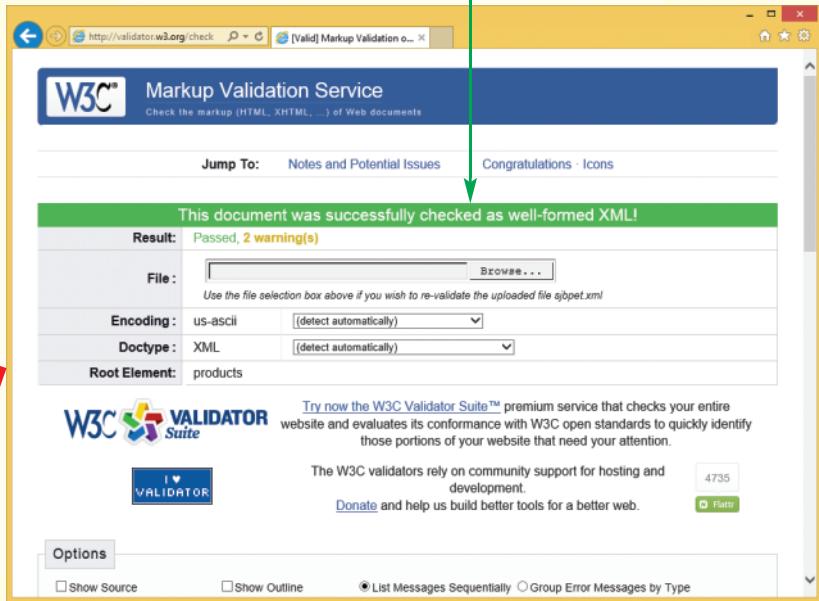
# Session 1.1 Visual Overview:



# XML Overview

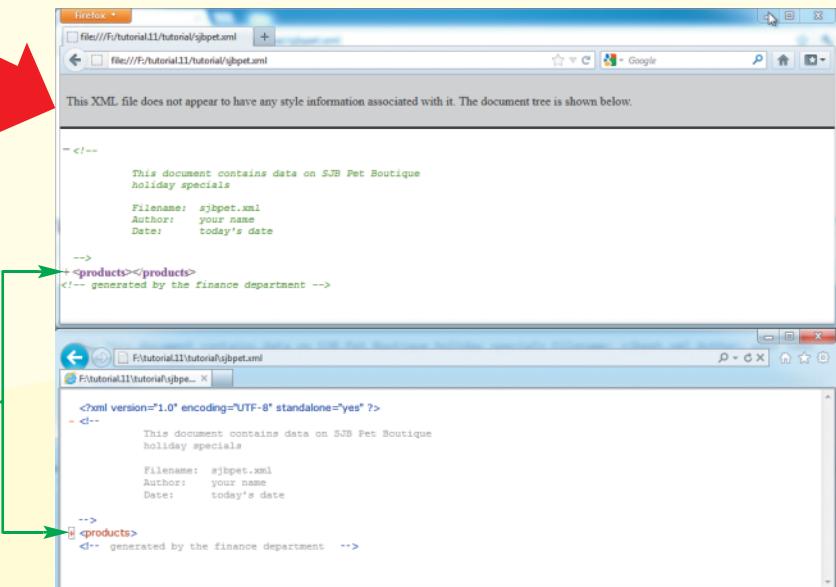
A **well-formed document** has no syntax errors and satisfies the general specifications for XML code defined by the World Wide Web Consortium (W3C).

An **XML parser** or **XML processor** interprets a document's code and verifies that it satisfies the W3C specifications.



## XML parser

Most major browsers display XML content in a hierarchical format by default.



## Introducing XML

The following short history lesson may help you better understand how XML fits in with today's technologies.

### The Roots of XML

XML has its roots in **Standard Generalized Markup Language (SGML)**, a language introduced in the 1980s that describes the structure and content of any machine-readable information. SGML is device-independent and system-independent. In theory, this means that documents written in SGML can be used on almost any type of device under almost any type of operating system. SGML has been the chosen vehicle for creating structured documents in businesses and government organizations of all sizes.

Even though SGML provides tools to manage enormous projects, it is a difficult language to learn and to apply because of its power, scope, and flexibility. XML can be thought of as a lightweight version of SGML. Like SGML, XML is a language used to create vocabularies for other markup languages, but it does not have SGML's complexity and expansiveness. XML is a markup language that is extensible, so it can be modified to match the needs of the document author and the data being recorded. The standards for XML are developed and maintained by the **World Wide Web Consortium (W3C)**, an organization created in 1994 to develop common protocols and standards for sharing information on the World Wide Web. When the W3C started planning XML, it established a number of design goals for the language. The syntax rules of XML are easy to learn and easy to use, as shown in Figure 1-1.

Figure 1-1

Highlights of XML syntax rules

Syntax Rule	Application
Every XML element must have a closing tag.	Every element must have a closing tag. A self-closing tag is permitted.
XML tags are case sensitive.	Opening and closing tags (or start and end tags) must be written with the same case.
XML elements must be properly nested.	All elements can have child (sub) elements. Child elements must be in pairs and be correctly nested within their respective parent element.
Every XML document must have a root element.	Every XML document must contain a single tag pair that defines the root element. All other elements must be nested within the root element.
XML elements can have attributes in name-value pairs.	Each attribute name within the same element can occur only once. Each attribute value must be quoted.
Some characters have a special meaning in XML.	The use of certain characters is restricted. If these characters are needed, entity references or character references may be used. References always begin with the character "&" (which is specially reserved) and end with the character ";".
XML allows for comments.	Comments cannot occur prior to the XML Declaration. Comments cannot be nested.

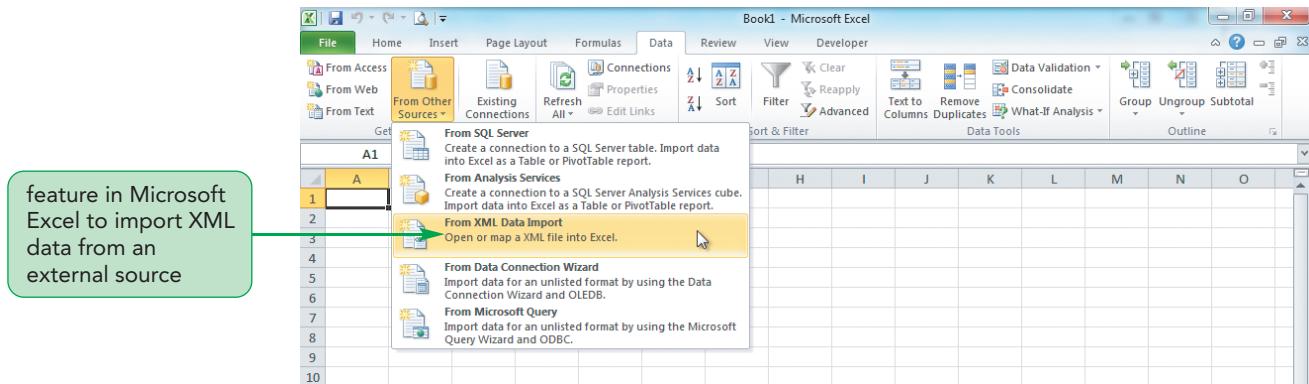
### XML Today

XML was originally created to structure, store, and transport information. Today, XML is still used for that purpose and has become the most common tool for data transmission among various applications. XML is used across a variety of industries, including accounting, banking, human resources, medical records, information technology, and insurance. Generally, it is used in all major websites, including major web services.

## XML with Software Applications and Languages

Currently, many software applications such as Microsoft Excel and Microsoft Word, and server languages such as Java, .NET, Perl, and PHP, can read and create XML files. As of the 2007 releases of Microsoft Office and OpenOffice, users can exchange data among Office applications and enterprise systems using XML and file compression technologies. Not only are the documents universally accessible, but the use of XML also reduces the risk of damaged files. Figure 1-2 shows Microsoft Excel's built-in mechanism for importing an XML file into an Excel spreadsheet.

**Figure 1-2 Importing XML data into Microsoft Excel**



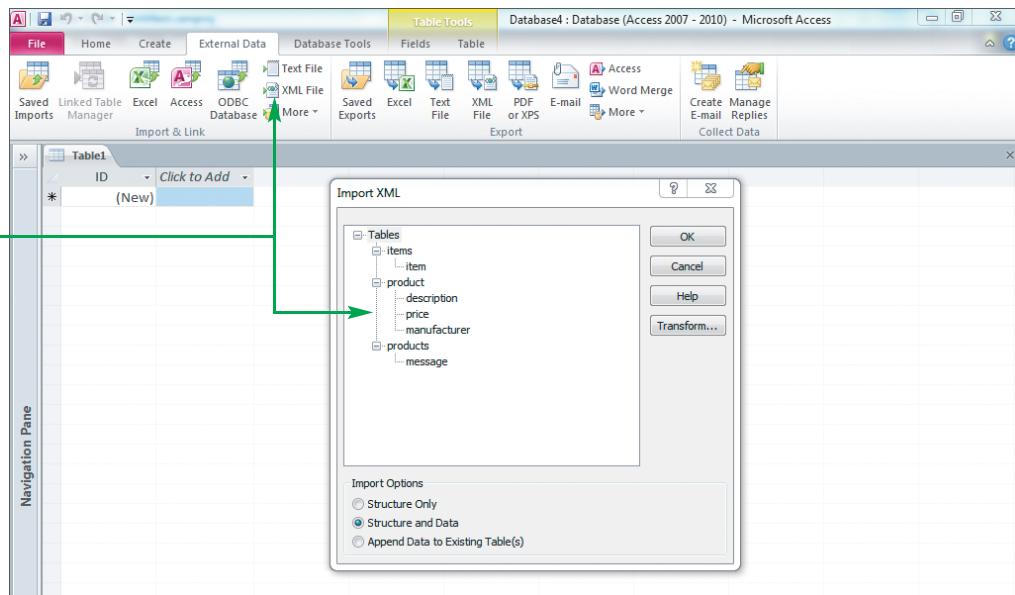
## XML and Databases

Databases store data, and XML is widely used for data interchange. All major databases, including Microsoft Access, Oracle, and MySQL, can read and create XML files. The fact that XML isn't platform-dependent gives the language flexibility as technologies change.

XML and relational databases are tightly woven together in most web applications. However, the two use distinctly different models to structure data. The relational model used by relational databases is based on two-dimensional tables, which have no hierarchy and no significant order. By contrast, XML is based on hierarchical trees in which order is significant. In the relational model, neither hierarchy nor sequence may be used to model information. In XML, hierarchy and sequence are the main methods used to represent information. This is one of the more fundamental differences between the two models, but there are more.

On web pages, XML is very useful because the structure of XML closely matches the structure used to display the same information in HTML. Both HTML and XML use tags in similar ways, often creating distinctly hierarchical structures to present data to users. Most of the data for web pages comes from relational databases and it must be converted to appropriate XML hierarchies for use in web pages. For these reasons, it makes more sense to see XML as a tool that works in conjunction with databases, rather than as a competitor to them. Major databases support easy-to-use integration with XML. For instance, Figure 1-3 shows how Access has incorporated easy XML importing and exporting of data.

Figure 1-3

**Importing XML-formatted data into Access****XML and Mobile Development**

It is highly doubtful that when members of the W3C got together to discuss XML, they even considered mobile device development and the importance that XML would play in this area. In fact, mobile device platforms such as Google’s Android and Apple’s iOS use XML in a variety of ways.

In iOS, Apple has built in the ability to import and export data classes in XML format. This makes it very easy to transfer information via XML. A popular use of XML in the iPhone is in a preference list or property list—commonly abbreviated as p-list—to organize data into named properties and lists of values, as shown in Figure 1-4.

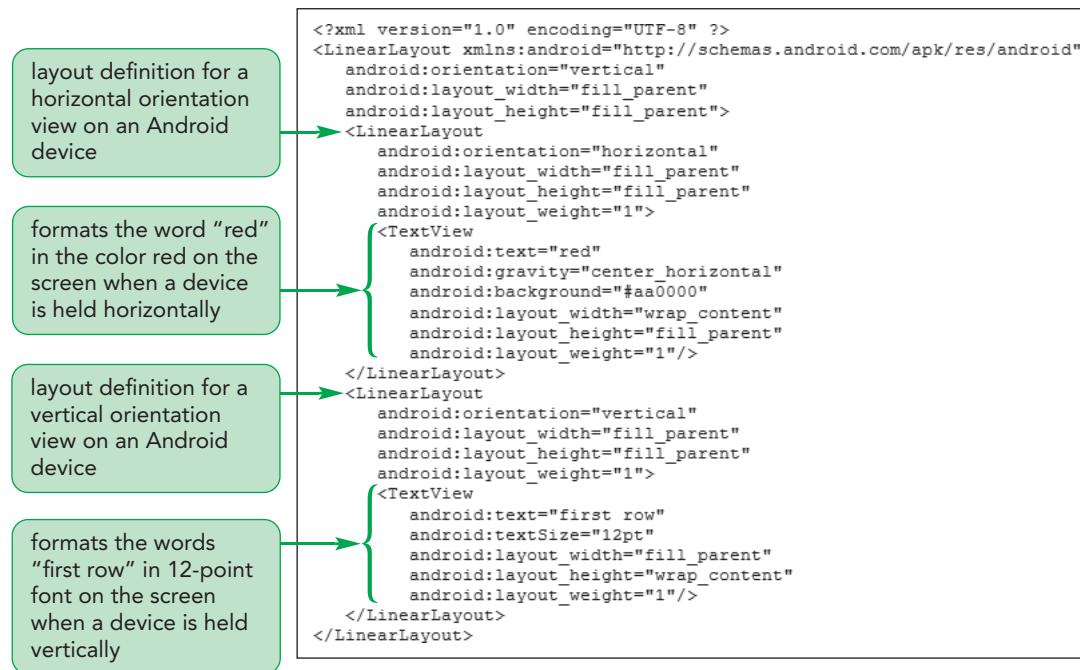
Figure 1-4

**iOS p-list file written in XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Accessories</key>
<string>Collar Tag</string>
<key>Shirt</key>
<string>Week Day</string>
<key>Bowls</key>
<string>Ceramic 2 Holder</string>
</dict>
</plist>
```

Android uses XML for screen layout and for working with data. Android provides a straightforward XML vocabulary for laying out content on the screen, allowing creation of XML layouts for different screen orientations, different device screen sizes, and different languages. Declaring an Android layout in XML makes it easier to visualize the structure of a user interface. Figure 1-5 shows an example of how Android uses XML to lay out the screen.

**Figure 1-5** Android layout definitions written in XML



## Creating an XML Vocabulary

HTML is an SGML application and is the foundation of web development. Like SGML, XML can be used to create **XML applications** or **vocabularies**, which are markup languages tailored to contain specific pieces of information. If Patricia wanted to create a vocabulary for the items in the SJB Pet Boutique product catalog, she might use XML to store the product information in the following format:

```

<productName>Dog Shirt Gift Basket</productName>
<manufacturer>SJB Pet Boutique</manufacturer>
<description>Something for every day of the week
</description>
<price currency="USD">$35.99</price>
<price currency="EUR">€26.79</price>
<productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206
</productItems>

```

You'll explore the structure and syntax of this document further in the next session, but you can already infer a lot about the type of information this document contains even without knowing much about XML. You can quickly see that this file contains data on a product named "Dog Shirt Gift Basket," including its manufacturer, its description, its two selling prices, and the product numbers of the items it includes.

The `productName`, `manufacturer`, `description`, `price`, and `productItems` elements in this example do not come from any particular XML specification; rather, they are custom elements that Patricia might create specifically for one of the SJB Pet Boutique documents.

Patricia could create additional elements describing things such as the product number, the seller name, and the quantity on hand. In this way, Patricia could create her own XML vocabulary that deals specifically with product, manufacturer, and inventory data.

You'll start your work for Patricia by examining an XML document and comparing the similarities between HTML and XML documents.

Like HTML documents, XML documents can be created and viewed with a basic text editor such as Notepad orTextEdit. More sophisticated XML editors are available, and using them can make it easier to design and test documents. However, you can complete the project in this tutorial with a basic text editor.

### To open an XML document in a text or XML editor:

- 1. In a text editor or XML editor, open **sjbCustomOrders.xml** from the **xml01 ▶ tutorial** folder where your data files are located. Figure 1-6 shows the contents of the sjbCustomOrders.xml document for the first order.

**Figure 1-6** Opening an XML document

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- This document contains data on SJB Pet Boutique
custom orders for the past week
-->
<customOrders>
    <order>
        <customer>
            <name>
                <firstName>John</firstName>
                <lastName>Taylor</lastName>
            </name>
            <address>
                <addressLine1>123 Main Street</addressLine1>
                <city>Monona</city>
                <state>WI</state>
                <zip>53716</zip>
            </address>
        </customer>
        <productNumber>12345</productNumber>
        <quantity>2</quantity>
        <unitPrice currency="USD">15.50</unitPrice>
        <salePrice currency="USD">25.00</salePrice>
    </order>

```

XML comments are just like those in HTML

opening and closing XML tags are just like those in HTML

XML attribute names and values are used to add additional information, just like in HTML

XML attribute values have quotes around them

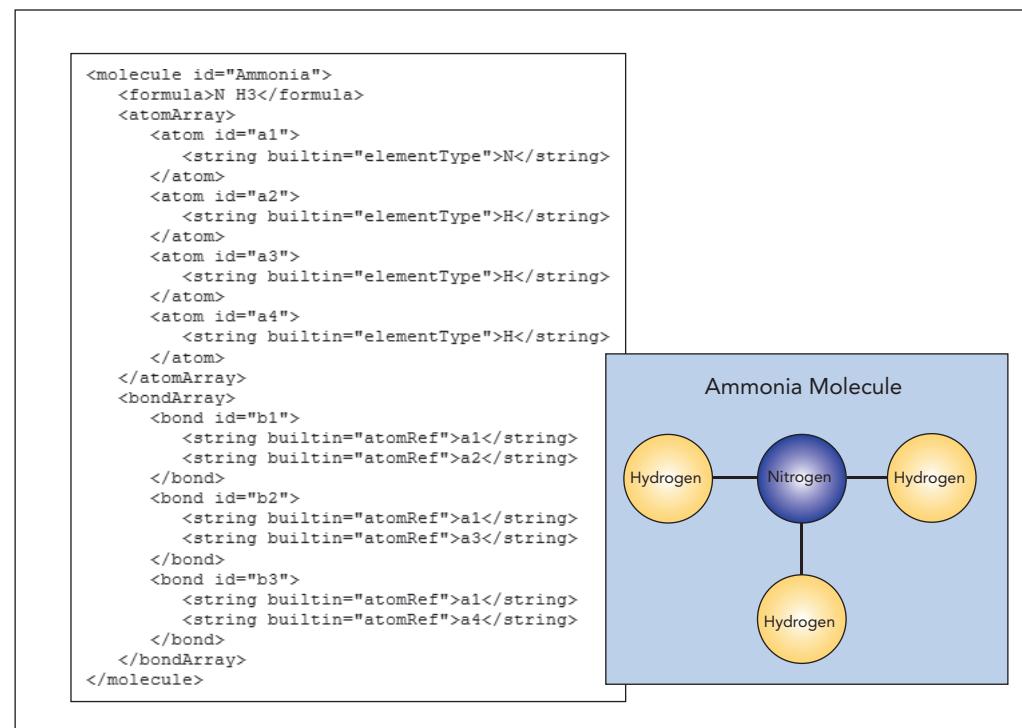
- 2. Examine the code, noting the similarities between an XML document and an HTML document, such as comments and opening and closing tags. You'll explore each aspect of an XML document's structure in the next session.
- 3. Close the file.

## Standard XML Vocabularies

If Patricia wanted to share the vocabulary that she uses for SJB Pet Boutique with other companies, she might use a standard vocabulary that is accepted throughout the industry. You can think of a **standard vocabulary** as a set of XML tags for a particular industry or business function. As XML has grown in popularity, standard vocabularies continue to be developed across a wide range of disciplines.

For example, chemists need to describe chemical structures containing hundreds of atoms bonded to other atoms and molecules. To meet this need, an XML vocabulary called **Chemical Markup Language (CML)** was developed, which codes molecular information. Figure 1-7 shows an example of a CML document used to store information on the ammonia molecule.

**Figure 1-7** Ammonia molecule described using CML



One of the more important XML vocabularies on the Internet is **Really Simple Syndication (RSS)**, which is the language used for distributing news articles or any content that changes on a regular basis. Subscribers to an RSS feed can receive periodic updates using a software program called a **feed reader** or an **aggregator**. Most current browsers contain some type of built-in feed reader to allow users to retrieve and view feeds from within the browser window. Most RSS feeds contain just links, headlines, or brief synopses of new information. Because an RSS file is written in XML, the RSS code follows the conventions of all XML documents. Figure 1-8 shows a segment of an RSS document.

**Figure 1-8** RSS document



Figure 1-9 lists a few of the many vocabularies that have been developed using XML.

**Figure 1-9**

**XML vocabularies**

XML Vocabulary	Description
Bioinformatic Sequence Markup Language (BSML)	Coding of bioinformatic data
Extensible Hypertext Markup Language (XHTML)	HTML written as an XML application
Mathematical Markup Language (MathML)	Presentation and evaluation of mathematical equations and operations
Music Markup Language (MML)	Display and organization of music notation and lyrics
Weather Observation Definition Format (OMF)	Distribution of weather observation reports, forecasts, and advisories
Really Simple Syndication (RSS)	Distribution of news headlines and syndicated columns
Synchronized Multimedia Integration Language (SMIL)	Editing of interactive audiovisual presentations involving streaming audio, video, text, and any other media type
Voice Extensible Markup Language (VoiceXML)	Creation of audio dialogues that feature synthesized speech, digitized audio, and speech recognition
Wireless Markup Language (WML)	Coding of information for smaller-screened devices, such as PDAs and cell phones

### TIP

You can learn more about several standard XML vocabularies at the W3C site, [www.w3.org/XML/](http://www.w3.org/XML/).

One of the more important XML vocabularies is **XHTML (Extensible Hypertext Markup Language)**, which is a reformulation of HTML as an XML application. You'll examine some properties of XHTML as you learn more about XML in the upcoming tutorials. Don't worry if you find all of these acronyms and languages a bit overwhelming.

## DTDs and Schemas

For different users to share a vocabulary effectively, rules must be developed that specifically control what code and content a document from that vocabulary might contain. This is done by attaching either a Document Type Definition (DTD) or a schema to the XML document containing the data. Both DTDs and schemas contain rules for how data in a document vocabulary should be structured. A DTD defines the structure of the data and, very broadly, the types of data allowable. A schema more precisely defines the structure of the data and specific data restrictions.

For example, Patricia can create a DTD or schema to require her documents to list the name, the manufacturer, a description, a list of prices, and a list of product items for each product in the SJB Pet Boutique inventory. DTDs and schemas are not required, but they can be quite helpful in ensuring that your XML documents follow a specific vocabulary. The standard vocabularies listed in Figure 1-9 all have DTDs to ensure that people in a given industry or area all work from the same guidelines.

To create a DTD or a schema, you simply need access to a basic text editor. You'll explore how to create DTDs and schemas in later tutorials.

## Well-Formed and Valid XML Documents

To ensure a document's compliance with XML rules, it can be tested against two standards—whether it's well formed, and whether it's valid. A well-formed document contains no syntax errors and satisfies the general specifications for XML code as laid out by the W3C. At a minimum, an XML document must be well formed or it will not be readable by programs that process XML code.

If an XML document is part of a vocabulary with a defined DTD or schema, it also must be tested to ensure that it satisfies the rules of that vocabulary. A well-formed XML document that satisfies the rules of a DTD or schema is said to be a **valid document**. In this tutorial, you'll look only at the basic syntax rules of XML to create well-formed documents. You'll learn how to test documents for validity in later tutorials.



### Problem Solving: Designing for Efficiency and Effectiveness

Although XML can do many different things, it is used most effectively to communicate data. In this respect, XML and databases go hand-in-hand—XML communicates data, and databases store data. XML delivers structured information in a generic format that's independent of how that information is used. As a result, the data does not rely on any particular programming language or software. XML developers have the freedom to work with a wide range of applications, devices, and complementary languages. A much larger benefit to the structural and logical markup is the ability to reuse portions of the information easily in any context where the information is structurally valid. Because XML focuses on communicating the data, the overall structure is simple and easy to design and maintain. This approach allows for a high level of efficiency and effectiveness, which in the long term reduces the amount of time and money spent on development and maintenance.

## Creating an XML Document

Now that you're familiar with the history and theory of XML, you're ready to create your first XML document.

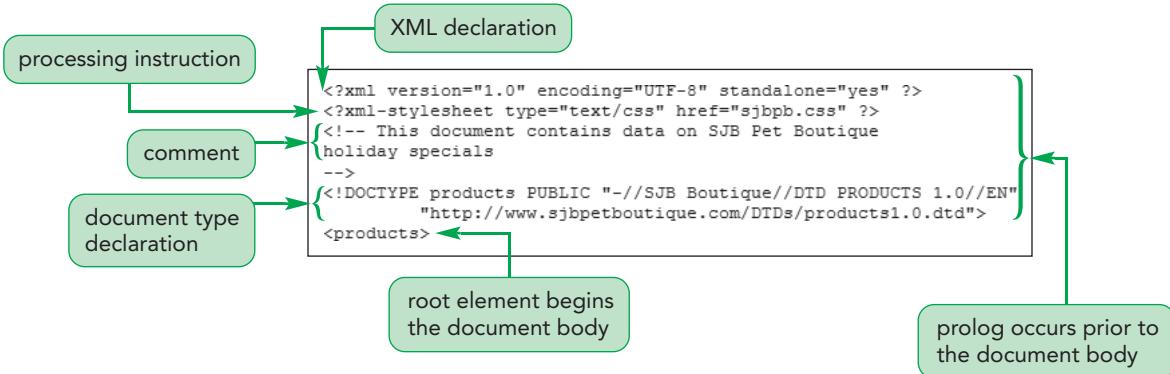
### The Structure of an XML Document

An XML document consists of three parts—the prolog, the document body, and the epilog. The prolog includes the following parts:

- **XML declaration:** indicates that the document is written in the XML language
- **Processing instructions** (optional): provide additional instructions to be run by programs that read the XML document
- **Comment lines** (optional): provide additional information about the document contents
- **Document type declaration (DTD)** (optional): provides information about the rules used in the XML document's vocabulary

Figure 1-10 illustrates the structure of a prolog.

**Figure 1-10 Structure of a prolog**



The document body, found immediately after the prolog, contains the document's content in a hierarchical tree structure. Following the document body is an optional epilog, which contains any final comments or processing instructions.

## The XML Declaration

The XML declaration is the first part of the prolog as well as the first line in an XML document. It signals to the program reading the file that the document is written in XML, and it provides information about how that code is to be interpreted by the program. The syntax of the XML declaration is

```
<?xml version="version number" encoding="encoding type"
       standalone="yes|no" ?>
```

where *version number* is the version of the XML specification being used in the document and *encoding type* identifies the character set used in the document. The default version value is 1.0. Although you can also specify a version value of 1.1, only a few programs support XML 1.1. With the growth of the web, XML 1.1 was implemented to allow almost any Unicode character to be used in object names.

**Unicode** is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's written languages. Unicode can be implemented using different character encodings; the most commonly used encodings are UTF-8 and UTF-16. Aside from forward compatibility with the Unicode standard in XML 1.1, there is not much difference between the 1.0 and 1.1 specifications, and most programmers still use the default version, 1.0. If you include the *standalone* or *encoding* attribute, you must include the *version* attribute.

Because different languages use different character sets, the *encoding* attribute allows XML to be used across a range of written languages. The default encoding scheme is UTF-8, which includes the characters used in almost all widely used written languages. However, a number of encoding schemes predate UTF-8, and some are still in use. If your XML document is intended for use with a system that uses a different encoding scheme, you might need to specify the scheme. For example, setting the *encoding* value to ISO-8859-1 tells a program reading the document that characters from the ISO-8859-1 (Latin-1) character set are being used in the document. The ISO-8859-1 character set has largely been replaced by UTF-8, but some systems and applications still use ISO-8859-1 encoding.

Finally, the *standalone* attribute indicates whether the document contains any references to external files. A *standalone* value of *yes* indicates that the document is self-contained, and a value of *no* indicates that the document requires additional information from external documents. The default value is *no*.

### *Creating an XML Declaration*

- To create an XML declaration, enter the code

```
<?xml ?>
```

in the first line of an XML document.

- To specify a version of XML to use, enter the code

```
version="version number"
```

after the opening `<?xml` tag, where `version number` is either 1.0 or 1.1.

- To specify a character encoding, enter the code

```
encoding="encoding type"
```

after the `version` attribute-value pair, where `encoding type` identifies the character set used in the document.

- To indicate whether the document is a standalone document, enter the code

```
standalone="yes | no"
```

after the `encoding` attribute-value pair, where the value `yes` or `no` indicates whether access to external files will be needed when processing the document.

Therefore, a sample XML declaration might appear as follows:

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

This declaration indicates that the XML version is 1.0, the ISO-8859-1 encoding scheme is being used, and the document is self-contained. If you instead entered the XML declaration

```
<?xml version="1.0" ?>
```

a processor would apply the default UTF-8 encoding scheme and the default `standalone` value of `no`.

Because XML is case sensitive, you cannot change the code to uppercase letters. The following code would generate an error because it is entered in uppercase.

#### **Not well-formed code:**

```
<?XML VERSION="1.0" ENCODING="ISO-8859-1" STANDALONE="YES" ?>
```

You also must ensure the quotation marks are included around the values in a declaration. The following XML declaration would result in an error because it is missing the quotation marks around the `version`, `encoding`, and `standalone` attribute values.

#### **Not well-formed code:**

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

The optional attributes for the XML declaration cannot be switched around. The following code would result in an error because the `standalone` attribute must come after the `encoding` attribute.

#### **Not well-formed code:**

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

The following statements are samples of well-formed options for coding an XML declaration:

### Well-formed code:

```
<?xml version="1.0" standalone="yes" ?>
<?xml version="1.1" standalone="no" ?>
<?xml version="1.0" encoding="UTF-8" ?>
<?xml version="1.1" encoding="UTF-8" ?>
```

Now that you've learned how to structure an XML declaration, you'll begin creating your first XML document by writing the prolog for an XML document to be used by SJB Pet Boutique.

### To create the basic structure of an XML document:

- 1. Use your text editor to open a blank document.
  - 2. Type the following line of code into your document:
- `<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>`
- 3. Press the **Enter** key, and then compare your code to Figure 1-11.

Be sure to include both question marks; otherwise, browsers will not recognize the declaration statement.

**Figure 1-11**

### Adding the XML declaration

The diagram shows a line of XML code: `<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>`. A green callout box points to the line with the text: "XML declaration must have both opening <? and closing ?> tags".

- 4. Save your document as **sjbpet.xml** in the `xml01 ▶ tutorial` folder.

**Trouble?** Some editors, such as Notepad, automatically assign the `.txt` extension to text files. To specify the `.xml` extension in Notepad, type `sjbpet.xml` in the File name box, click All Files from the Save as type list box, and then click the Save button. If you're using a different text editor, consult that program's documentation.

## Inserting Comments

Patricia wants you to include information in the document about its purpose and contents. Comments are one way of doing this. They can appear anywhere in the prolog after the XML declaration. Comments in the prolog provide additional information about what the document will be used for and how it was created. Generally speaking, comments are ignored by programs reading the document and do not affect the document's contents or structure.

To insert a comment in an XML document, enter

```
<!-- comment -->
```

where `comment` is the text of the comment. Comments cannot be placed before the XML declaration and cannot be embedded within tags or other comments. You should avoid using the two dashes (--) anywhere in a comment except at the beginning and the end. If you have a comment that will occupy more than one line, you can continue the comment on as many lines as you need. All text after `<!--` is considered a comment until the close of the comment tag, which is signaled by the two dashes. Anything within comments is effectively invisible, so only code outside of the comments needs to be valid and well formed.

You'll add comments to the `sjbpet.xml` file describing its contents and purpose.

### To insert an XML comment:

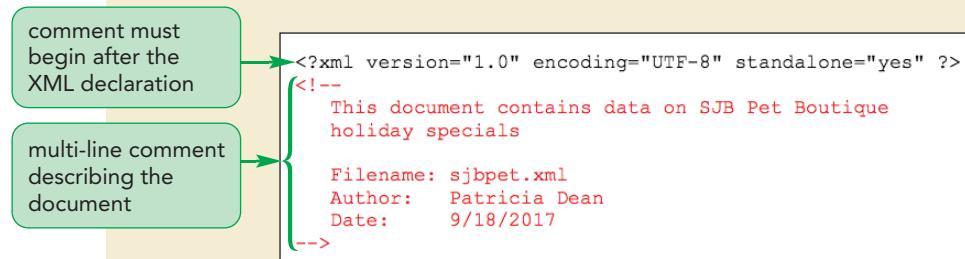
- 1. In the sjbpet.xml file, enter the following comment lines directly below the XML declaration, pressing the **Enter** key after each line, replacing the text *your name* and today's date with your name and the current date, respectively, and indenting as shown:

```
<!--
    This document contains data on SJB Pet Boutique
    holiday specials

    Filename: sjbpet.xml
    Author:   your name
    Date:     today's date
-->
```

Figure 1-12 shows the new comment in the document.

**Figure 1-12** Adding a comment to the XML document



- 2. Directly below the comment closing tag, add the following code:

```
<products>
    <!-- document body content -->
</products>
```

This code serves as a placeholder for the document body.

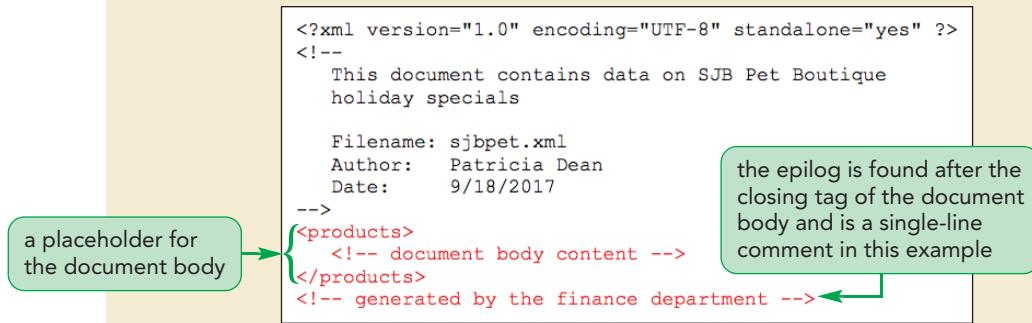
**Trouble?** Some XML editors automatically add the closing HTML tag after you type the starting tag. Be sure to type the element content before the closing tag.

- 3. Add a comment to the document's epilog by placing the following text directly below the closing `</products>` tag:

```
<!-- generated by the finance department -->
```

Figure 1-13 shows the new placeholder and comment tags in the document.

**Figure 1-13** Adding a comment to the epilog



- 4. Save your changes to the sjbpet.xml document.

Patricia is pleased that you've created a basic XML document so quickly. Next, you'll add the content.

**INSIGHT**

### Commenting an XML Document

Although a developer should easily be able to read and understand the code of an XML document, it's good practice to include comments in a document's prolog. Typically, these comments summarize the contents of the XML data. If there's anything special about the original creation of the XML document, that information should be included as well.

Some businesses have very specific standards for how comments should be coded within a document. These rules might even dictate where opening and closing tags for a comment are coded. For example, you might encounter coding standards that require single-line and multi-line comments to have the opening tag, comments, and closing tag on separate lines, as in the following code:

```
<!--  
    single-line comment or multi-line comment  
-->
```

Likewise, you might see coding standards that allow a single-line comment to have the opening and closing tags on the same line, as follows:

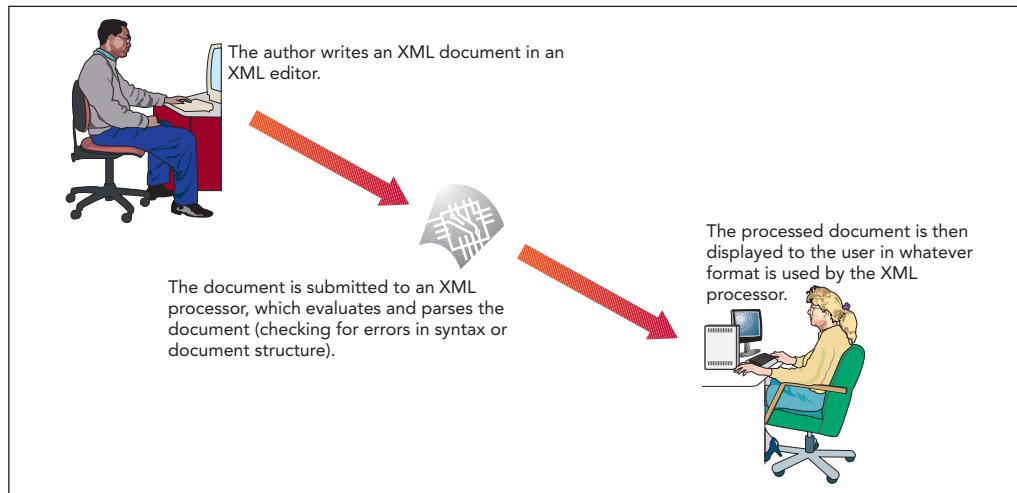
```
<!-- single-line comment -->
```

## Processing an XML Document

Now that you've created the very basic content for Patricia's pet store document, you can work with browsers to display that content.

### XML Parsers

A program that reads and interprets an XML document is called an XML processor or XML parser, or simply a **processor** or **parser**. A parser has several functions. First, a parser interprets a document's code and verifies that it satisfies all the XML specifications for document structure and syntax. XML parsers are strict. If even one tag is omitted or one lowercase character should be uppercase, a parser reports an error and rejects the document. This might seem excessive, but that rigidity was built into XML to eliminate viewers' ability to interpret how the code is displayed—much like HTML gives web browsers wide discretion in interpreting markup code. A second function of a parser is to interpret PCDATA in a document and resolve any character or entity references found within the document. Finally, an XML document might contain processing instructions that tell a parser exactly how the document should be read and interpreted. A third job of a parser is to interpret these instructions and carry them out. Figure 1-14 illustrates the parsing process from document creation to final presentation.

**Figure 1-14****XML parsing process****TIP**

Because XML, by definition, includes no predefined elements, browsers can't apply default styles to elements as they do to HTML content; instead, they default to showing the document's tree structure.

The current versions of all major web browsers include an XML parser of some type.

When an XML document is submitted to a browser, the XML parser built into the browser first checks for syntax errors. If it finds none, the browser then displays the contents of the document. Older browsers might display only the data content; newer browsers display both the data and the document's tree structure. Current versions of Internet Explorer, Chrome, and Firefox display XML documents in an expandable/collapsible outline format that allows users to hide nested elements. This is supported by a built-in extensible style sheet, which you'll learn more about in later tutorials. The various parts of a document might be color coded within the browser, making the document easier to read and interpret. Opera displays raw XML that is not expandable/collapsible, but is color coded. Safari parses the XML; in order to view the raw XML in Safari, you must select the View Source command.

You're ready to test whether the sjbpet.xml document is well formed. To test for well-formedness, you'll use an XML parser to compare the XML document against the rules established by the W3C. The web has many excellent sources for parsers that check for well-formedness, including websites to which you can upload an XML document. Several editors check XML code for well-formedness as well.

*NOTE: The following steps instruct you to upload your document to validator.w3.org to check Patricia's sjbpet.xml document for well-formedness. If you don't have Internet access and you're using an XML editor, consult that program's documentation and follow those steps to check your document for well-formedness.*

**To check the sjbpet.xml file for well-formedness:**

- 1. In your web browser, open [validator.w3.org](http://validator.w3.org), and then click the **Validate by File Upload** tab. The W3C Markup Validation Service page opens, as shown in Figure 1-15.

Figure 1-15

## W3C Markup Validation Service

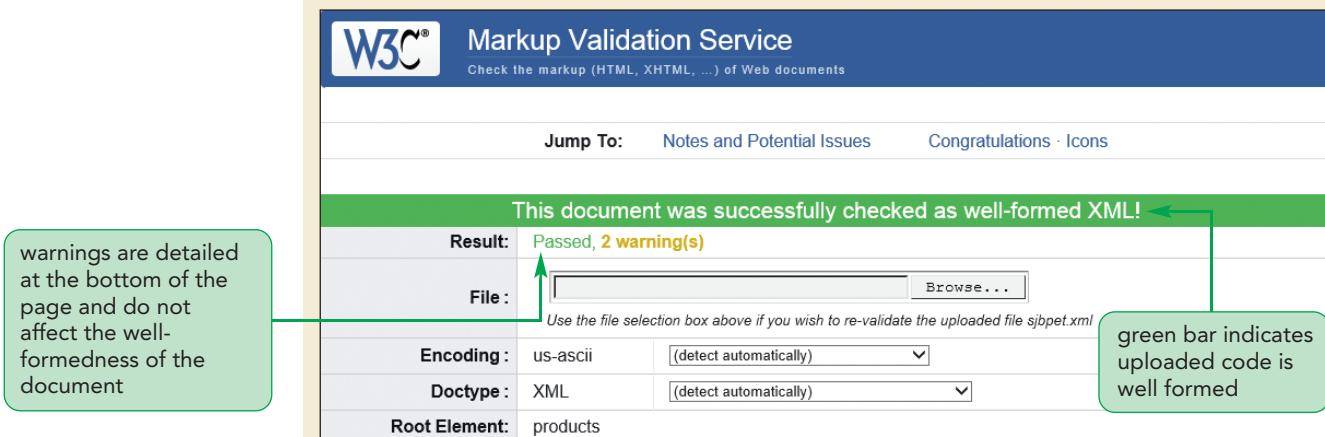


Copyright © 2014 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University, Beihang). All Rights Reserved. <http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231>

- 2. Click the **Browse** or **Choose File** button, in the dialog box that opens navigate to your **xml01 > tutorial** folder, click **sjbpet.xml** in the file list, and then click **Open**.  
**Trouble?** If the file list does not display the .xml extension at the end of the filename, select the file named sjbpet with a type of XML File.
- 3. Click the **Check** button. The file you selected is uploaded, and then the validation service displays your validation results, as shown in Figure 1-16.

Figure 1-16

## Results of check for well-formedness



Copyright © 2014 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University, Beihang). All Rights Reserved. <http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231>

**Trouble?** If the results page shows a red bar and indicates that the document was not found to be well-formed XML, check your XML code against the sjbpet.xml code shown in Figure 1-13. Your code must match exactly, including the use of uppercase and lowercase letters. Fix any discrepancies, be sure to save your changes, and then repeat Steps 1-3.

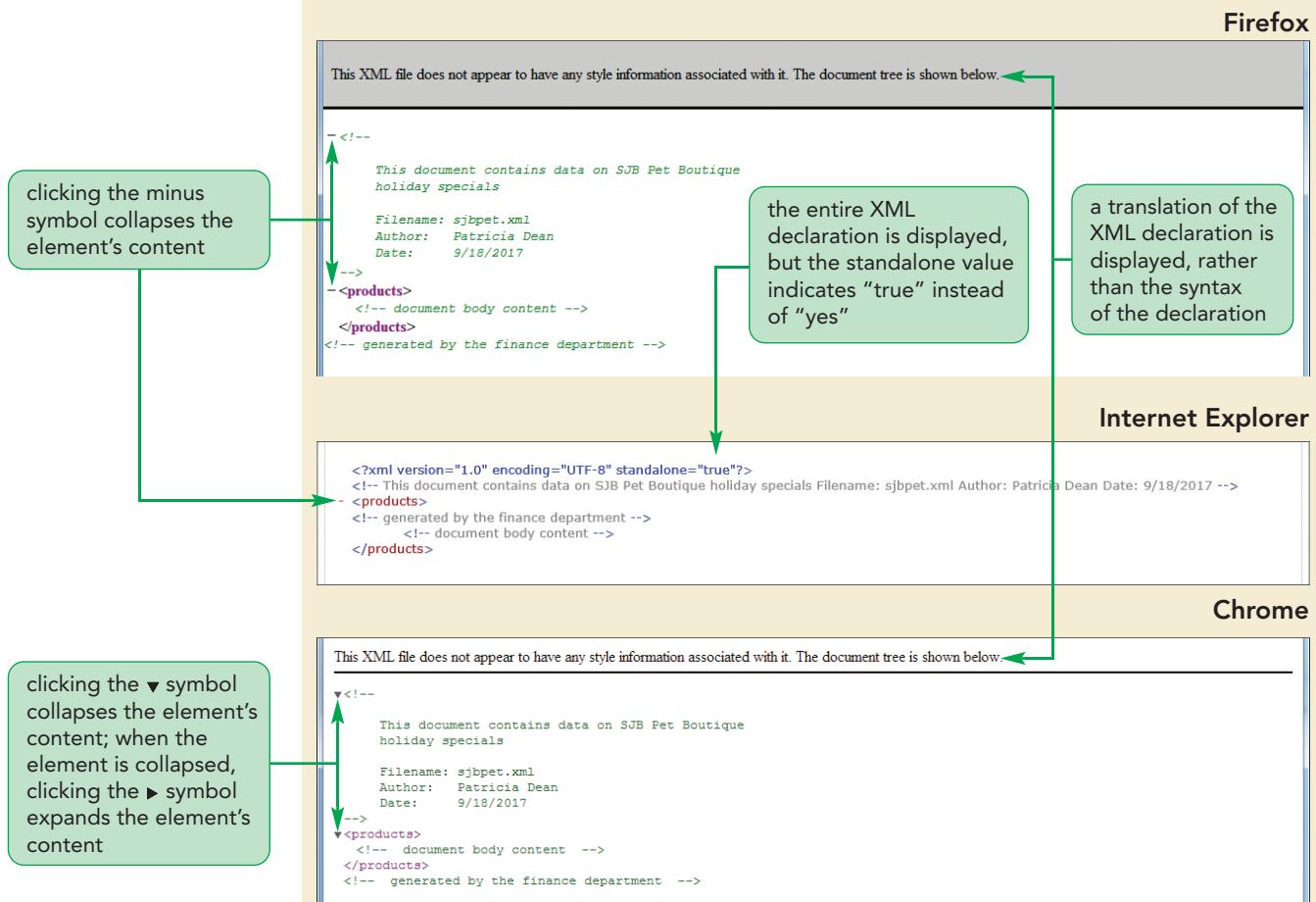
Most web browsers also function as XML parsers. To see how Patricia's basic document is displayed, you'll open it now in your browser.

### To view the sjbpet.xml file in your browser:

- 1. Open the **sjbpet.xml** document in your web browser. Different browsers display XML content differently. Figure 1-17 shows the contents of the file as it appears in current versions of Firefox, Internet Explorer, and Chrome at the time this book was published. In Safari, the window is blank because the browser displays only the content of an XML document, and your document is currently empty.

Figure 1-17

Displaying the sjbpet.xml document in Firefox, Internet Explorer, and Chrome



**Trouble?** If Internet Explorer displays a yellow information bar, click the bar, select Allow Blocked Content, and then click Yes in the Security Warning dialog box to fully display the file.

**Trouble?** If the sjbpet.xml file opened in another application instead of your browser, XML documents are probably associated with that application on your system. To instead open an XML document in your browser, locate the file in your file manager, right-click the filename, point to Open with, and then click the name of your browser in the list of available programs.

**Trouble?** In Windows 8, Internet Explorer may not be configured to open XML files stored locally. If you're using Windows 8, you may need to use a different browser, such as Firefox or Chrome, to complete the steps in this tutorial.

- 2. If you are running a browser that displays the contents of the document in outline form, click the **minus (-)** or the **down-pointing triangle (▼)** in front of the `<products>` tag. The browser collapses the comment nested within the `products` element.
- 3. Click the **plus (+)** or the **right-pointing triangle (►)** in front of the `<products>` tag. The browser expands the comment nested within the `products` element.

Because the XML file is well formed, the browser has no trouble rendering the document content. Patricia now wants to see how browsers respond to an XML document that is not well formed. She asks you to intentionally introduce an error into the `sjbpet.xml` file to verify that the error is flagged by the browser.

#### To see the result of an error in the `sjbpet.xml` file:

- 1. Return to the `sjbpet.xml` document in your XML editor.
  - 2. Change the second to the last line of the file from `</products>` to `</PRODUCTS>` and then save the changes to your document.
- Once you make this change, your document is no longer well formed because element names are case sensitive and a closing tag must match its opening tag.
- 3. In your web browser, open [validator.w3.org](http://validator.w3.org), and then click the **Validate by File Upload** tab.
  - 4. Click the **Browse** or **Choose File** button, navigate to your `xml01 > tutorial` folder, click `sjbpet.xml` in the file list, click **Open**, and then click **Check**. The page now displays a red bar near the top, indicating that errors were found.
  - 5. Scroll down to the Validation Output section of the page. As Figure 1-18 shows, the page displays two errors and a third informational message.

Figure 1-18

#### Errors from a document that is not well formed

The screenshot shows the 'Validation Output' section of the W3C Validator. It displays three items:

- Validation Output: 2 Errors**
  - Line 12, Column 11: end tag for element "PRODUCTS" which is not open**  
`</PRODUCTS>`  
 The Validator found an end tag for the above element, but that element is not currently open. This is often caused by a leftover end tag from an element that was removed during editing, or by an implicitly closed element (if you have an error related to an element being used where it is not allowed, this is almost certainly the case). In the latter case this error will disappear as soon as you fix the original problem.  
If this error occurred in a script section of your document, you should probably read this [FAQ entry](#).
  - Line 13, Column 45: end tag for "products" omitted, but OMITTAG NO was specified**  
`<!-- generated by the finance department -->`  
 You may have neglected to close an element, or perhaps you meant to "self-close" an element, that is, ending it with ">" instead of "<".
- Line 10, Column 1: start tag was here**  
`<products>`

A green callout box points to the first error message with the text: "errors indicate two unmatched tags". Another green callout box points to the third item with the text: "informational message describes the location of the unmatched start tag".

Copyright © 2014 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University, Beihang). All Rights Reserved. <http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231>

- 6. Refresh or reload the `sjbpet.xml` document in your web browser. Figure 1-19 shows the contents of the file as it appears in Firefox, Chrome, and Safari.

Figure 1-19

**Firefox, Chrome, and Safari renderings of a document that is not well formed**

The figure shows three browser windows side-by-side. The top window is Firefox, which displays a detailed error message: "XML Parsing Error: mismatched tag. Expected: </products>. Location: file:///E:/xml01/tutorial/sjbpet.xml Line Number 12, Column 3:" followed by the XML code "</PRODUCTS>". A callout box on the left says "Firefox displays helpful information about an XML parsing error". The middle window is Chrome and the bottom window is Safari, both showing a summary of the error: "This page contains the following errors: error on line 12 at column 12: Opening and ending tag mismatch: products line 0 and PRODUCTS Below is a rendering of the page up to the first error." A callout box on the left says "Chrome and Safari display helpful information about the error, but also provide the raw XML".

**Trouble?** Recent versions of Internet Explorer handle case-sensitivity of XML and CSS differently from all other browsers and show a blank window. To display the error message in Internet Explorer, press the F12 key to open the developer tools if they are not open at the bottom of the window, click Browser Mode on the Developer Tools menu bar, and then click Internet Explorer 8. The error message should now appear.

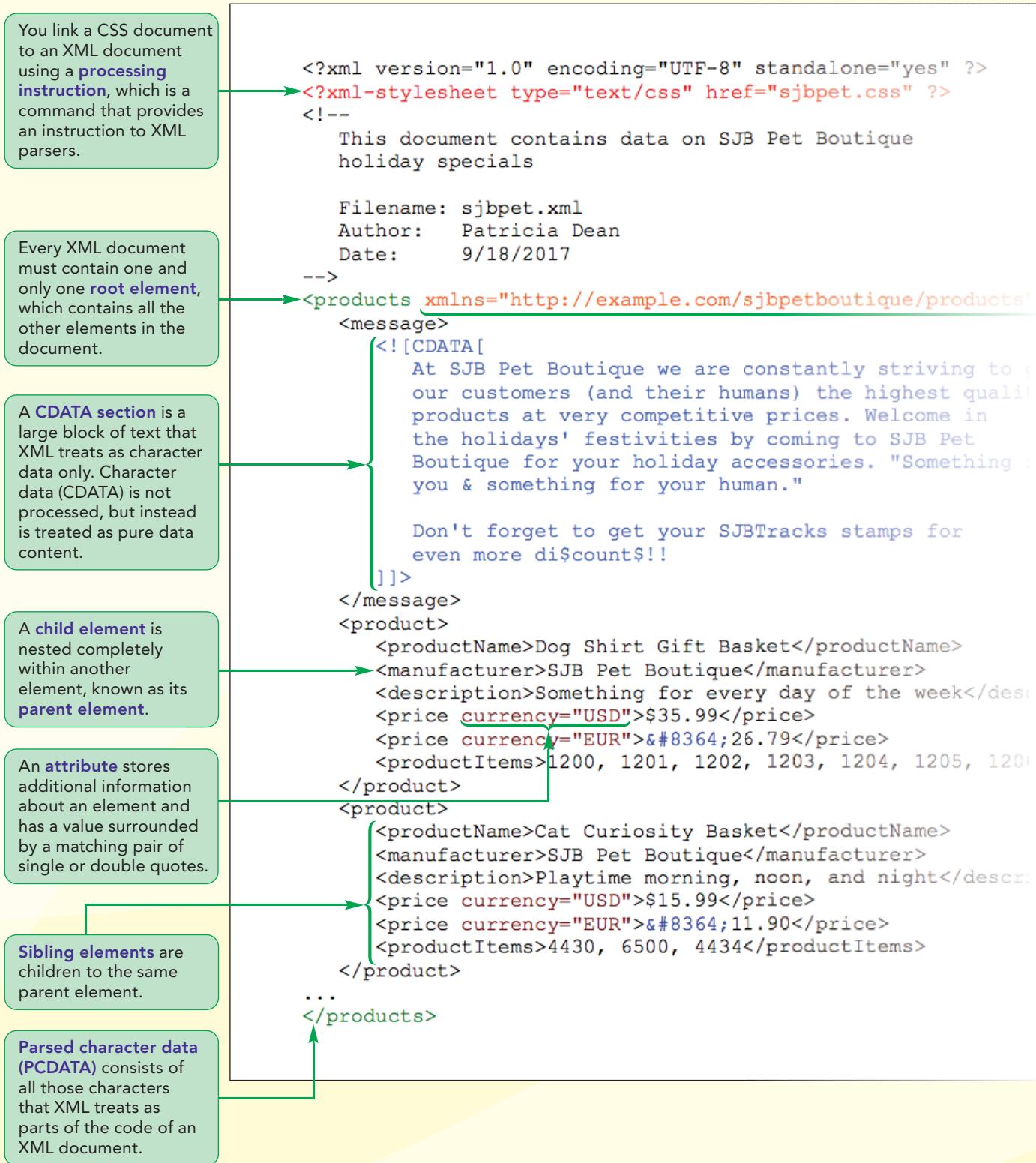
- 7. Return to your XML editor and then replace </PRODUCTS> with </products>. This returns the code to its original, well-formed state.
- 8. Save your changes to the document, repeat Steps 3 and 4, and then verify that the error message is replaced with an indication that the document is well formed.
- 9. Refresh or reload the **sjbpet.xml** document in your web browser, and then verify that the browser displays the document contents without any errors.

You have created a well-formed XML document containing a prolog, the shell for the document body, and an epilog. In the next session, you'll focus your attention on adding the contents of the document body.

**REVIEW****Session 1.1 Quick Check**

1. Define the term “extensible.” How does the concept of extensibility relate to XML?
2. Name three limitations of HTML that led to the development of XML.
3. What is an XML vocabulary? Provide an example of an XML vocabulary.
4. What are the three parts of an XML document?
5. Provide an XML declaration that specifies that the document supports XML version 1.0, uses the ISO-8859-1 encoding scheme, and does not require information from other documents.
6. Write the XML code that creates a comment with the text “Data extracted from the sjbpetboutique.com database”.
7. What is an XML parser?
8. What happens if you attempt to open an XML document in an XML parser when that document contains syntax errors?

# Session 1.2 Visual Overview:



# Structuring an XML Document

```
.0" encoding="UTF-8" standalone="yes" ?>
  type="text/css" href="sjbpet.css" ?>

contains data on SJB Pet Boutique
als

pet.xml
ricia Dean
8/2017

<a href="http://example.com/sjbpetboutique/products">
  <!-- Pet Boutique we are constantly striving to give
      customers (and their humans) the highest quality
      products at very competitive prices. Welcome in
      today's festivities by coming to SJB Pet
      Boutique for your holiday accessories. "Something for
      something for your human." -->
  <p>Don't forget to get your SJBTracks stamps for
      a discount!!</p>

  <!-- Product items -->
  <productItems>
    <item>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price currency="USD">$35.99</price>
    <price currency="EUR">€26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </productItems>

  <productItems>
    <item>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night</description>
    <price currency="USD">$15.99</price>
    <price currency="EUR">€11.90</price>
    <productItems>4430, 6500, 4434</productItems>
  </productItems>
</a>
```

A **Uniform Resource Identifier (URI)** is a text string that uniquely identifies a resource; one version of a URI is the **Uniform Resource Locator (URL)**, which is used to identify the location of a resource on the web, and the other is a **Uniform Resource Name (URN)**, which provides a persistent name for a resource, independent of that resource's location.

A **namespace** is a defined collection of element and attribute names.

**Character references** in XML work the same as character references in HTML.

## Working with Elements

The document body in an XML document is made up of elements that contain data to be stored in the document. **Elements** are the basic building blocks of XML files. An element can have text content and child element content. The content is stored between an **opening tag** and a **closing tag**, just as in HTML. The syntax of an XML element with text content is

```
<element>content</element>
```

where *element* is the name given to the element and *content* represents the text content of the element. The opening tag is `<element>`, and `</element>` is the closing tag. Element names usually are selected by XML authors to describe element contents. Element names might be established already if an author is using a particular XML vocabulary, such as VoiceXML. As you saw in the last session, Patricia can store the name of a manufacturer using the following line of code:

```
<manufacturer>SJB Pet Boutique</manufacturer>
```

There are a few important points to remember about XML elements:

- Element names are case sensitive, which means that, for example, `itemnumber`, `itemNumber`, and `ItemNumber` are unique elements.
- Element names must begin with a letter or the underscore character ( \_ ) and may not contain blank spaces. Thus, you cannot name an element `Item Number`, but you can name it `Item_Number`.
- Element names cannot begin with the string `xml` because that group of characters is reserved for special XML commands.
- The name in an element's closing tag must exactly match the name in the opening tag.
- Element names can be used more than once, so the element names can mean different things at different points in the hierarchy of an XML document.

The following element text would result in an error because the opening tag is capitalized and the closing tag is not, meaning that they are not recognized as the opening and closing tags for the same element.

### Not well-formed code:

```
<MANUFACTURER>SJB Pet Boutique</manufacturer>
```

### REFERENCE

#### Creating XML Elements

- To create an XML element, use the syntax

```
<element>content</element>
```

where *element* is the name given to the element, *content* represents the text content of the element, `<element>` is the opening tag, and `</element>` is the closing tag.

- To create an empty XML element with a single tag, use the following syntax:

```
<element />
```

- To create an empty XML element with a pair of tags, use the syntax

```
<element></element>
```

## Empty Elements

Not all elements contain content. An **open element** or **empty element** is an element with no content. White space, such as multiple spaces or a tab, is not considered content. An empty element tag usually is entered using a **one-sided tag** that obeys the syntax

```
<element />
```

where *element* is the name of the empty element. Alternatively, you can enter an empty element as a two-sided tag with no content, as follows:

```
<element></element>
```

Most programmers prefer the one-sided tag syntax to avoid confusion with two-sided tags, which normally have content.

All of the following are examples of empty elements:

```
<sample1 />
<sample2></sample2>
<sample3> </sample3>
<sample4>

</sample4>
```

Empty XML elements are similar to HTML's collection of empty elements, such as the `<br />` tag for line breaks or the `<img />` tag for inline graphics.

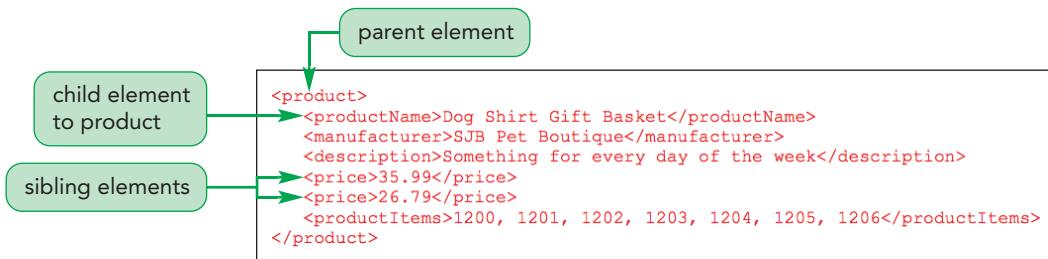
If empty elements have no content, why use them in an XML document? One reason is to mark certain sections of the document for programs reading it. For example, Patricia might use an empty element to distinguish one group of products from another. Empty elements can also contain attributes that can be used to store information. Finally, empty elements can be used to reference external documents containing non-textual data in much the same way that the HTML `<img />` tag is used to reference image files.

## Nesting Elements

In addition to text content, elements also can contain other elements. An element contained within another element is said to be a **nested** element. For instance, in the following example, multiple elements are nested within the `product` element:

```
<product>
  <productName>Dog Shirt Gift Basket</productName>
  <manufacturer>SJB Pet Boutique</manufacturer>
  <description>Something for every day of the week</description>
  <price>35.99</price>
  <price>26.79</price>
  <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206
  </productItems>
</product>
```

Like HTML, XML uses familial names to refer to the hierarchical relationships between elements. A nested element is a child element of the element in which it is nested—its parent element. Elements that are side-by-side in a document's hierarchy are sibling elements. In the example in Figure 1-20, the `productName`, `manufacturer`, `description`, `price`, and `productItems` elements are siblings to each other, and each of these elements is also a child of the `product` element.

**Figure 1-20** Parent, child, and sibling elements

A common syntax error in creating an XML document is improperly nesting one element within another. Just as in XHTML, XML does not allow the opening and closing tags of parent and child elements to overlap. For this reason, the following XML code is not considered well formed because the `productName` element does not completely enclose the `manufacturer` element.

#### **Not well-formed code:**

```

<productName>Dog Shirt Gift Basket<manufacturer>SJB Pet Boutique
</productName></manufacturer>

```

To make the code well formed, the closing `</productName>` tag should be moved after the `</manufacturer>` tag to prevent any overlap of the element tags, as follows:

#### **Well-formed code:**

```

<productName>Dog Shirt Gift Basket<manufacturer>SJB Pet Boutique
</manufacturer></productName>

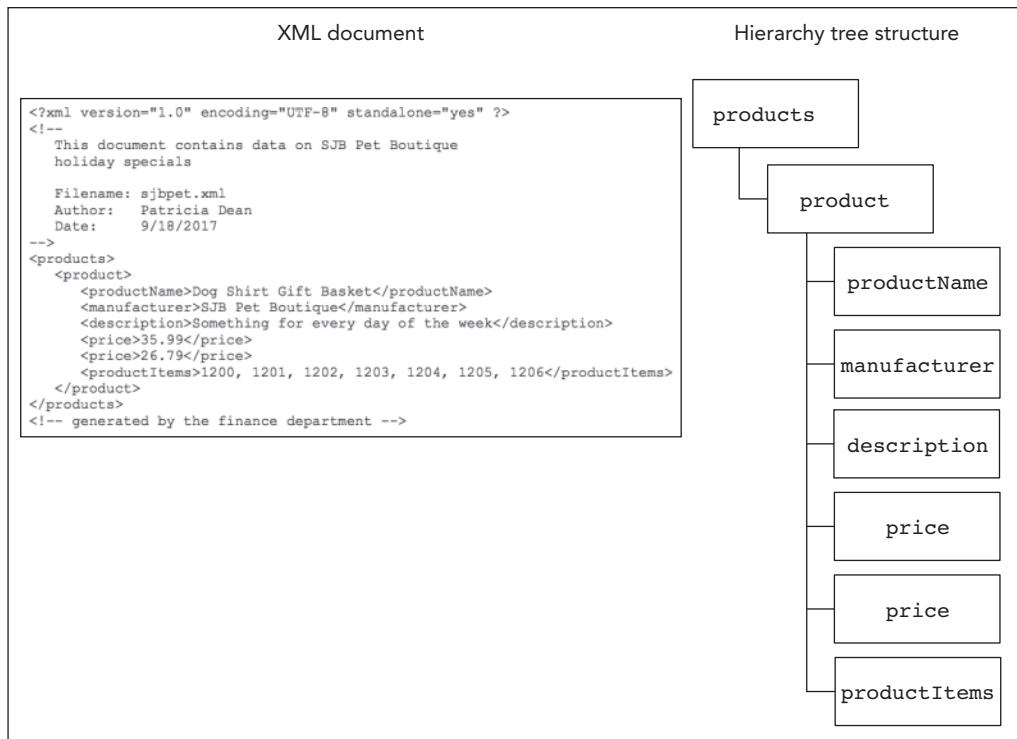
```

This syntax is correct because both `manufacturer` tags are within the opening and closing tags of the parent `productName` element.

## The Element Hierarchy

The familial relationship of parent, child, and sibling extends throughout the entire document body. All elements in the body are children of a single element called the root element or **document element**. Figure 1-21 shows a sample XML document with its hierarchy represented in a tree diagram. The root element in this document is the `products` element. Note that the XML declaration and comments are not included in the tree structure of the document body.

**Figure 1-21** Code for an XML document along with its corresponding tree diagram



An XML document must include a root element to be considered well formed. The following document code is not well formed because it lacks a single root element containing all other elements in the document body.

#### Not well-formed code:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!--
    This document contains data on SJB Pet Boutique
    holiday specials
-->
<productName>Dog Shirt Gift Basket</productName>
<manufacturer>SJB Pet Boutique</manufacturer>
<description>Something for every day of the week</description>
<price>35.99</price>
<price>26.79</price>
<productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206
</productItems>

```



PROSKILLS

### Written Communication: Writing Code Visually

Including comments in a file is one way of communicating information about the file contents to other developers. The way you visually arrange your code is another way to communicate with other developers. Technically, child and sibling elements do not have to be coded on separate lines; parsers do not care. Thus, the following code, while challenging to understand at a glance, is considered well formed:

```
<product><productName>Dog Shirt Gift Basket</productName>
<manufacturer>SJB Pet Boutique</manufacturer><description>
Something for every day of the week</description><price>35.99
</price><price>26.79</price><productItems>1200, 1201, 1202,
1203, 1204, 1205, 1206</productItems></product>
```

However, by indenting the code and placing siblings on their own lines, you can visually reveal the hierarchy relationships and add a dimension of visual communication to your code. The following code is the same as the previous example, but now line breaks and indents have been added:

```
<product>
  <productName>Dog Shirt Gift Basket</productName>
  <manufacturer>SJB Pet Boutique</manufacturer>
  <description>Something for every day of the week
  </description>
  <price>35.99</price>
  <price>26.79</price>
  <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206
  </productItems>
</product>
```

Although the elements are coded exactly the same in both instances, a programmer would spend a lot more time identifying the parent, child, and sibling relationships in the first coding sample. By including simple line breaks and tabs in your documents, you can communicate the structure of your document in a way that complements written comments.

## Charting the Element Hierarchy

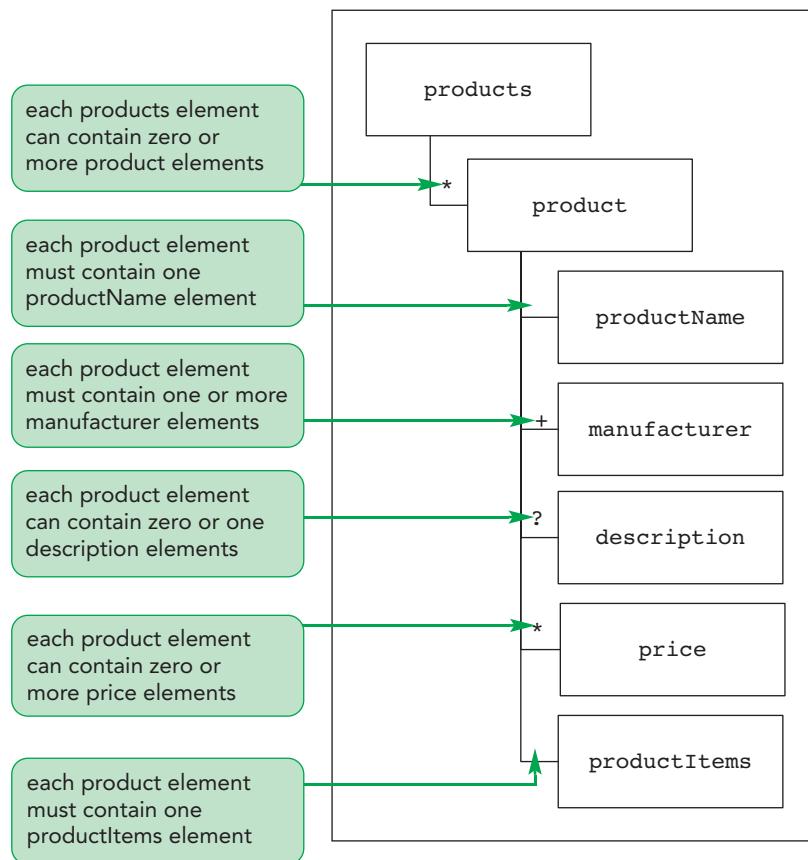
A quick way to view the overall structure of a document body is to chart the elements in a tree structure like the one shown in Figure 1-21. This can become confusing, however, when a single element has several children of the same type. For example, the `product` element in Figure 1-21 contains two `price` elements within it. Other product descriptions might have differing numbers of `price` elements using this layout. It would be useful to have a general tree diagram that indicates whether a particular child element can occur zero times, once, or several times within a parent. Figure 1-22 displays the shorthand code you will see in the tree diagrams in this and subsequent tutorials to indicate the general structure of the document body.

**Figure 1-22** Charting the number of child elements

Symbol	Description	Chart	Interpretation
[none]	The parent contains a single occurrence of the child element.	<pre> graph TD     product[product] --- productName[productName]   </pre>	A product element must contain a single productName element.
?	The parent contains zero or one of the child elements.	<pre> graph TD     product[product] -- "?" --&gt; description[description]   </pre>	A product element may contain a description element.
*	The parent contains zero or more of the child elements.	<pre> graph TD     product[product] ---* manufacturer[manufacturer]   </pre>	A product element can contain zero or more manufacturer elements.
+	The parent contains at least one of the child elements.	<pre> graph TD     product[product] ---+ price[price]   </pre>	A product element must contain one or more price elements.

Figure 1-23 shows how these symbols apply to the structure of the XML document you will create for SJB Pet Boutique.

**Figure 1-23** Charting the sjbpet.xml document



The symbols ?, \*, and + are part of the code used in creating DTDs to validate XML documents. Using these symbols in a tree diagram prepares you to learn more about DTDs later on.

## Writing the Document Body

Now that you've reviewed some of the aspects of XML elements, you're ready to use them in an XML document. You've already started creating the sjbpet.xml document, which will describe the SJB Pet Boutique's holiday specials. Patricia would like you to add information on the products shown in Figure 1-24 to this document. Patricia has indicated that no current XML vocabulary exists that meets her data requirements. Because you'll create your own XML vocabulary for this document, you'll use descriptive element names for all of the element items.

Figure 1-24

**Product descriptions for the SJB Pet Boutique holiday specials**

<b>Product Name</b>	<b>Manufacturer</b>	<b>Description</b>	<b>Price</b>	<b>Product Items</b>
Dog Shirt Gift Basket	SJB Pet Boutique	Something for every day of the week	35.99 26.79	1200, 1201, 1202, 1203, 1204, 1205, 1206
Cat Curiosity Basket	SJB Pet Boutique	Playtime morning, noon, and night	15.99 11.90	4430, 6500, 4434
Piggy Snuggle Basket	ACME		17.50 13.03	3230, 3232
Dog Snuggle Basket	ACME		14.25 10.61	3230, 3232, 3250

The `products` element will be the root element of the document. The `productName`, `manufacturer`, `description`, `price`, and `productItems` elements all will be children of a parent `product` element for each item.

**To add the products to the XML document:**

- 1. If you took a break after the previous session, make sure the **sjbpet.xml** document is open in your editor.
- 2. Delete the code `<!-- document body content -->` from the document. You'll replace it with the actual document body content.
- 3. Between the opening and closing `<products>` tags, insert the following code, pressing the **Enter** key at the end of each line and indenting as shown:

```

<product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week
    </description>
    <price>35.99</price>
    <price>26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206
    </productItems>
</product>
<product>
    <productName>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night
    </description>
    <price>15.99</price>
    <price>11.90</price>
    <productItems>4430, 6500, 4434</productItems>
</product>
<product>
    <productName>Piggy Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price>17.50</price>
    <price>13.03</price>
    <productItems>3230, 3232</productItems>
</product>

```

```
<product>
  <productName>Dog Snuggle Basket</productName>
  <manufacturer>ACME</manufacturer>
  <price>14.25</price>
  <price>10.61</price>
  <productItems>3230, 3232, 3250</productItems>
</product>
```

Figure 1-25 shows the code for the products added to the document.

**Figure 1-25** Adding elements to the XML document body

The diagram illustrates the structure of the XML code. A green callout points to the opening tag '<products>' with the text 'root element begins the document body'. Another green callout points to the first child element of the 'products' element with the text 'child elements indented within the product element to make the code easier to read'. The XML code itself is as follows:

```
<products>
  <product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price>35.99</price>
    <price>26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </product>
  <product>
    <productName>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night</description>
    <price>15.99</price>
    <price>11.90</price>
    <productItems>4430, 6500, 4434</productItems>
  </product>
  <product>
    <productName>Piggy Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price>17.50</price>
    <price>13.03</price>
    <productItems>3230, 3232</productItems>
  </product>
  <product>
    <productName>Dog Snuffle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price>14.25</price>
    <price>10.61</price>
    <productItems>3230, 3232, 3250</productItems>
  </product>
</products>
```

- 4. Save your changes to the **sjbpet.xml** document, and then use your editor or [validator.w3.org](http://validator.w3.org) to check your document for well-formedness.

**Trouble?** If your document is no longer well formed, check each line of code associated with an error message against the code in Step 3 and fix any errors. Save your work and, if necessary, repeat until the document is well formed.

You've finished adding the product elements and their child elements. Next, you'll add attributes to the sjbpet.xml document.

## Working with Attributes

Every element in an XML document can contain one or more attributes. An attribute describes a feature or characteristic of an element. The syntax for adding an attribute to an element is

```
<element attribute="value"> ... </element>
```

where **attribute** is the attribute's name and **value** is the attribute's value. In the case of a one-sided element tag, the syntax is as follows:

```
<element attribute="value" />
```

Attribute values are text strings. Therefore, an attribute value always must be enclosed within either single or double quotes. For example, if Patricia wants to include the currency as an attribute of the **price** element, she could enter the following code:

```
<price currency="USD">35.99</price>
```

Alternatively, she could instead use single quotes, as follows:

```
<price currency='USD'>35.99</price>
```

Because they're considered text strings, attribute values may contain spaces and almost any character other than the less than ( < ) and greater than ( > ) symbols. You can choose any name for an attribute as long as it meets the following rules:

- An attribute name must begin with a letter or an underscore ( \_ ).
- Spaces are not allowed in attribute names.
- Like an element name, an attribute name should not begin with the text string *xml*.

An attribute name can appear only once within an element. Like all other aspects of XML, attribute names are case sensitive, and incorrect case is a common syntax error found in attributes. An attribute named **Currency** is considered distinct from an attribute named **currency**, so it's important to be consistent in your case when naming attributes.

## REFERENCE

### *Adding an Attribute to an Element*

- To add an attribute to an element, use the syntax

```
<element attribute="value"> ... </element>
```

where *element* is the name given to the element, *attribute* is the attribute's name, and *value* is the attribute's value.

- To add an attribute to a single-sided tag, use the syntax

```
<element attribute="value" />
```

- To specify multiple attributes for a single element, use the syntax

```
<element attribute1="value1" attribute2="value2" ...> ... </element>
```

where *attribute1* is the first attribute's name, *value1* is the first attribute's value, *attribute2* is the second attribute's name, *value2* is the second attribute's value, and so on. Each attribute is separated by a space.

Most of SJB Pet Boutique's business is in the U.S., but the company also has a sizable customer base in Ireland. For this reason, the XML document includes the prices for each item both in the U.S. dollar and in Ireland's currency, the euro. Patricia would like to specify the currency as an attribute of each **price** element instead of as a separate element. She wants to use the abbreviation **USD** for prices in U.S. dollars and **EUR** for prices in euros. Figure 1-26 shows the currency for each **price** element in the **sjbpet.xml** document.

**Figure 1-26**      **Currency values for price elements**

Product Name	Price	Currency
Dog Shirt Gift Basket	35.99	USD
	26.79	EUR
Cat Curiosity Basket	15.99	USD
	11.90	EUR
Piggy Snuggle Basket	17.50	USD
	13.03	EUR
Dog Snuggle Basket	14.25	USD
	10.61	EUR

You'll modify the sjbpet.xml document to include a **currency** attribute with the proper value for each **price** element.

**TIP**

Make sure you enclose the value of the attribute within double quotes.

**To add the currency attribute to each price element:**

- 1. Return to the **sjbpet.xml** document in your editor.
- 2. Locate the first **price** element for the first product, which has a value of 35.99.
- 3. Directly before the **>** in the **price** element's opening tag, insert a space and then type **currency="USD"**.
- 4. Repeat Steps 2 and 3 to add **currency** attributes to the remaining **price** elements, using the values found in Figure 1-26. Figure 1-27 shows the updated code.

**Figure 1-27****Number attributes added to document body**

attribute name and value are defined in the opening tag of each element

```

<products>
  <product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price currency="USD">35.99</price>
    <price currency="EUR">26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </product>
  <product>
    <productName>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night</description>
    <price currency="USD">15.99</price>
    <price currency="EUR">11.90</price>
    <productItems>40, 6500, 4434</productItems>
  </product>
  <product>
    <productName>Piggy Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">17.50</price>
    <price currency="EUR">13.03</price>
    <productItems>3230, 3232</productItems>
  </product>
  <product>
    <productName>Dog Snugglu Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">14.25</price>
    <price currency="EUR">10.61</price>
    <productItems>3230, 3232, 3250</productItems>
  </product>
</products>

```

- 5. Save the changes to your document.

**INSIGHT****Elements vs. Attributes**

It's not always clear when to use an attribute value rather than inserting a new element. For example, the following code provides currency information about each `price` element using an attribute:

```
<price currency="USD">35.99</price>
<price currency="EUR">26.79</price>
```

However, you could instead insert the currency information for each `price` element as a sibling element using the following format:

```
<price>35.99</price>
<currency>USD</currency>
<price>26.79</price>
<currency>EUR</currency>
```

Either style is acceptable. Some developers argue that attributes never should be used because they increase a document's complexity. Their rationale is that when information is placed as an element rather than as an attribute within an element, it's more easily accessible by programs reading a document.

A general rule of thumb is that if all of the XML tags and their attributes were removed from a document, the remaining text should comprise the document's content or information. In this scenario, if an attribute value is something you want displayed, it really should be placed in an element, as in the second example above. However, if an attribute is not necessary in order to understand the document content, you can safely keep it as an attribute placed within an element, as in the first example above.

Another rule of thumb is that attributes should be used to describe data, but should not contain data themselves. However, this can be a difficult distinction to make in most cases.

So which should you use? Different developers have different preferences, and there's no right answer. However, when you're in doubt, it's probably safest to use an element.

## Using Character and Entity References

Next, Patricia would like you to make sure that the symbols for the item prices display in browsers. Figure 1-28 displays the prices of the four products with the corresponding currency symbols.

Figure 1-28

Product prices

Product	U.S. Price	Ireland Price
Dog Shirt Gift Basket	\$35.99	€26.79
Cat Curiosity Basket	\$15.99	€11.90
Piggy Snuggle Basket	\$17.50	€13.03
Dog Snuffle Basket	\$14.25	€10.61

To insert characters such as the € symbol, which is not available on a standard U.S. keyboard, you use a **numeric character reference**, also known simply as a **character reference**. The syntax for a character reference is

`&#nnn;`

**TIP**

Unicode can be used in conjunction with entity references to display specific characters or symbols.

where *nnn* is a character number from the ISO/IEC character set. The **ISO/IEC character set** is an international numbering system for referencing characters from virtually any language. Character references in XML work the same as character references in HTML.

Because it can be difficult to remember the character numbers for different symbols, some symbols also can be identified using a **character entity reference**—also known simply as an **entity reference**—using the syntax

`&entity;`

where *entity* is the name assigned to the symbol.

Make sure to start every character or entity reference with an ampersand ( & ) and end with a semicolon ( ; ). Figure 1-29 lists a few of the commonly used character and entity references.

Figure 1-29

Character and entity references

Symbol	Character Reference	Entity Reference	Description
>	&#62;	&gt;	Greater than
<	&#60;	&lt;	Less than
'	&#27;	&apos;	Apostrophe (single quote)
"	&#22;	&quot;	Double quote
&	&#38;	&amp;	Ampersand
©	&#169;	&copy;	Copyright
®	&#174;	&reg;	Registered trademark
™	&#153;		Trademark
°	&#176;		Degree
£	&#163;		Pound
€	&#8364;	&euro;	Euro
¥	&#165;	&yen;	Yen

Notice that not all symbols have both character and entity references.

**REFERENCE****Inserting Character and Entity References**

- To insert a character reference into an XML document, use

`&#nnn;`

where *nnn* is a character reference number from the ISO/IEC character set.

- To insert an entity reference, use

`&entity;`

where *entity* is a recognized entity name.

A common mistake made in XML documents is to forget that XML processors interpret the ampersand symbol ( & ) as the start of a character reference and not as a character. Often, XML validators catch such mistakes. For example, the following code results in an error message because the ampersand symbol is not followed by a recognized character reference number or entity name.

**Not well-formed code:**

```
<manufacturer>Hills & Barton</manufacturer>
```

To avoid this error, you need to use the &#38; character reference or the &amp; entity reference in place of the ampersand symbol, as follows:

**Well-formed code:**

```
<manufacturer>Hills &amp; Barton</manufacturer>
```

Character references are sometimes used to store the text of HTML code within an XML element. For example, to store the HTML tag `` in an element named `htmlCode`, you need to use character references &#60; and &#62; to reference the < and > symbols contained in the HTML tag. The following code accomplishes this:

```
<htmlCode>&#60;img src="sjblogo.jpg" /&#62;</htmlCode>
```

The following code would not give the same result.

**Not well-formed code:**

```
<htmlCode></htmlCode>
```

When encountering this code, an XML processor would attempt to interpret `` as an empty element within the document and not as the content of the `htmlCode` element.

The character reference for the € symbol is &#8364;. You'll use this character reference now to add Ireland's currency symbol to the second `price` element for each product. You'll also use the keyboard to add the \$ symbol to the first `price` element for each product.

**To insert the dollar symbol and the euro character reference into the price elements:**

- 1. Return to the `sjbpet.xml` document in your text editor.
- 2. Within the first `product` element, click after the opening tag for the first `price` element, and then type the \$ character.
- 3. Within the first `product` element, click after the opening tag for the second `price` element, and then type &#8364;.
- 4. Repeat Steps 2 and 3 for each of the remaining three `product` elements, as shown in Figure 1-30. You should insert both the \$ character and the character reference a total of four times within the document.

**TIP**

After typing the first occurrence of the character reference, you can copy the reference to the Clipboard and then paste it in the other three locations to minimize typing errors.

Figure 1-30

## \$ and euro symbol character reference inserted

character reference starts with & and ends with ;

```
<products>
  <product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price currency="USD">$35.99</price>
    <price currency="EUR">&#8364;26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </product>
  <product>
    <productName>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night</description>
    <price currency="USD">$15.99</price>
    <price currency="EUR">&#8364;11.90</price>
    <productItems>4430, 6500, 4434</productItems>
  </product>
  <product>
    <productName>Piggy Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">$17.50</price>
    <price currency="EUR">&#8364;13.03</price>
    <productItems>3230, 3232</productItems>
  </product>
  <product>
    <productName>Dog Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">$14.25</price>
    <price currency="EUR">&#8364;10.61</price>
    <productItems>3230, 3232, 3250</productItems>
  </product>
</products>
```

- 5. Save your changes to the document.
- 6. If necessary, start your web browser and then open the **sjbpet.xml** document. As Figure 1-31 shows, each occurrence of the &#8364; character reference is converted into a € symbol.

Figure 1-31

## € symbol rendered in browser

browser replaces each occurrence of &#8364; with € symbol

```
-<products>
  -<product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price currency="USD">$35.99</price>
    <price currency="EUR">€26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </product>
  -<product>
    <productName>Cat Curiosity Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Playtime morning, noon, and night</description>
    <price currency="USD">$15.99</price>
    <price currency="EUR">€11.90</price>
    <productItems>4430, 6500, 4434</productItems>
  </product>
  <product>
    <productName>Piggy Snuggle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">$17.50</price>
    <price currency="EUR">€13.03</price>
    <productItems>3230, 3232</productItems>
  </product>
  <product>
    <productName>Dog Snuffle Basket</productName>
    <manufacturer>ACME</manufacturer>
    <price currency="USD">$14.25</price>
    <price currency="EUR">€10.61</price>
    <productItems>3230, 3232, 3250</productItems>
  </product>
</products>
```

## Understanding Text Characters and White Space

As you've seen from working on the sjbpet.xml document, XML documents consist only of text characters. However, text characters fall into three categories—parsed character data, character data, and white space. In order to appreciate how programs like browsers interpret XML documents, it's important to understand the distinctions among these categories.

### Parsed Character Data

Parsed character data (PCDATA) consists of all those characters that XML treats as parts of the code of an XML document. This includes characters found in the following:

- the XML declaration
- the opening and closing tags of an element
- empty element tags
- character or entity references
- comments

Parsed character data also can be found in other XML features that you'll learn about in later tutorials, such as processing instructions and document type declarations.

The presence of PCDATA can cause unexpected errors to occur within a document. XML treats any element content as potential PCDATA because elements may contain other elements. This means that symbols such as &, <, or >, which are all used in creating markup tags or entity references, are extracted and the appropriate content is used in your program. For example, the following line would result in an error because the greater than symbol (>) in the temperature value would be viewed as the end of a markup tag.

#### Not well-formed code:

```
<temperature> > 98.6 degrees</temperature>
```

Because the greater than symbol does not have any accompanying markup tag characters, the document would be rejected for being not well formed. To correct this error, you would have to replace the greater than symbol with either the character reference &#62; or the entity reference &gt;. The correct temperature value would then be entered as follows:

#### Well-formed code:

```
<temperature>&gt;98.6 degrees</temperature>
```

If you instead wanted to display the above line without it being parsed, you would code it as follows:

```
&lt;temperature&gt; &gt;98.6 degrees&lt;/temperature&gt;
```

## Character Data

After you account for parsed character data, the remaining symbols constitute a document's actual contents, known as **character data**. Character data is not processed, but instead is treated as pure data content. One purpose of character and entity references is to convert PCDATA into character data. For example, when the program reading an XML document encounters the entity reference &gt;, it converts it to the corresponding character data symbol >.

## White Space

The third type of character that an XML document can contain is white space.

**White space** refers to nonprintable characters such as spaces (created by pressing the spacebar), new line characters (created by pressing the Enter key), or tab characters (created by pressing the Tab key). Processors reading an XML document must determine whether white space represents actual content or is used to make the code more readable. For example, the code you've entered in the SJB Pet Boutique document is indented to make it more readable to users. However, this does not have any impact on how the XML document's contents or structure are interpreted.

HTML applies **white space stripping**, in which consecutive occurrences of white space are treated as a single space. White space stripping allows HTML authors to format a document's code to be readable without affecting the document's appearance in browsers. As a result of white space stripping, the HTML code

```
<p>This is a  
paragraph.</p>
```

is treated the same as

```
<p>This is a paragraph.</p>
```

White space is treated slightly differently in XML. Technically, no white space stripping occurs for element content, which means that the content of the XML element

```
<paragraph>This is a  
paragraph.</paragraph>
```

is interpreted as

```
This is a  
paragraph.
```

This preserves both the new line character and all of the blank spaces.

However, the majority of browsers today transform XML code into HTML, and in the process apply white space stripping to element content. Thus, in browsers such as Internet Explorer, Firefox, or Chrome, the contents of the above XML element would be displayed as

*This is a paragraph.*

When white space appears in places other than element content, XML treats it in the following manner:

- White space is ignored when it is the only character data between element tags; this allows XML authors to format their documents to be readable without affecting the content or structure.
- White space is ignored within a document's prolog and epilog, and within any element tags.
- White space within an attribute value is treated as part of the attribute value.

In summary, white space in an XML document is generally ignored unless it is part of the document's data.

### TIP

Not all browsers parse white space in the same way, so be sure to preview your documents in a variety of browsers.

## Creating a CDATA Section

Sometimes an XML document needs to store significant blocks of text containing the < and > symbols. For example, what if you wanted to place all of the text from this tutorial into an XML document? If there were only a few < and > symbols, it might not be too much work to replace them all with &#60; and &#62; character references, or with &lt; and &gt; entity references. However, given the volume of text in this tutorial, it would be cumbersome to replace all of the < and > symbols with the associated character or entity references, and the code itself would be difficult to read.

As an alternative to using character references, you can place text into a CDATA section. A CDATA section is a block of text that XML treats as character data only. The syntax for creating a CDATA section is

```
<![CDATA [  
    character data  
]]>
```

A CDATA section may contain most markup characters, such as <, >, and &. In a CDATA section, these characters are interpreted by XML parsers or XML editors as text rather than as markup commands. A CDATA section

- may be placed anywhere within a document.
- cannot be nested within other CDATA sections.
- cannot be empty.

The only sequence of symbols that may not occur within a CDATA section is ]] because this is the marker ending a CDATA section.

The following example shows an element named `htmlCode` containing a CDATA section used to store several HTML tags:

```
<htmlCode>  
    <![CDATA[  
        <h1>SJB Pet Boutique</h1>  
        <h2>Fashion for Pets and Their Humans</h2>  
    ]]>  
</htmlCode>
```

The text in this example is treated by XML as character data, not PCDATA. Therefore, a processor would not read the `<h1>` and `<h2>` character strings as element tags. You might find it useful to place any large block of text within a CDATA section to protect yourself from inadvertently inserting a character such as the ampersand symbol that would be misinterpreted by an XML processor.

Patricia would like you to insert a message element into the `sjbpet.xml` document that describes the purpose and contents of the document. You'll use a CDATA section for this task.

### To create a CDATA section:

- 1. Return to the `sjbpet.xml` document in your XML editor.
- 2. Insert a blank line below the opening `<products>` tag, and then enter the following code, pressing **Enter** at the end of each line:

```
<message>  
    <![CDATA[  
        At SJB Pet Boutique we are constantly striving to give  
        our customers (and their humans) the highest quality  
        products at very competitive prices. Welcome in  
        the holidays' festivities by coming to SJB Pet  
        Boutique for your holiday accessories. "Something for  
        you & something for your human."  
  
        Don't forget to get your SJBTTracks stamps for  
        even more di$count$!!  
    ]]>  
</message>
```

Figure 1-32 shows the code for the CDATA section in the document.

Figure 1-32

### Adding a CDATA section

The screenshot shows an XML document structure. A green callout points to the opening CDATA tag `<![CDATA[` with the text "opening CDATA tag". Another green callout points to the closing CDATA tag `]]&gt;` with the text "closing CDATA tag". A third green callout on the right side of the code area contains the text "press the Enter key at the end of each line to match this text wrap". The XML code is as follows:</p>

```
-->
<products>
  <message>
    <![CDATA[
      At SJB Pet Boutique we are constantly striving to give
      our customers (and their humans) the highest quality
      products at very competitive prices. Welcome in
      the holidays' festivities by coming to SJB Pet
      Boutique for your holiday accessories. "Something for
      you & something for your human."
      Don't forget to get your SJBTTracks stamps for
      even more di$count$!!
    ]]>
  </message>
  <product>
    <productName>Dog Shirt Gift Basket</productName>
```

- 3. Save the changes to your document.
- 4. Verify that your document is well formed, and correct any errors if necessary.

**Trouble?** If your code is not well formed, check your XML code against the sjbpet.xml code shown in Figure 1-32. Your newly added code should match exactly, including the use of uppercase and lowercase letters. Fix any discrepancies, be sure to save your changes, and then verify that your document is well formed.

You've completed your work to create Patricia's XML document. Next, you'll display the document in a web browser.

**INSIGHT**

### CDATA Cans and Cannots

New authors of XML documents often mistakenly use CDATA to protect data from being treated as ordinary character data during processing. Character data is character data, regardless of whether it is expressed via a CDATA section or ordinary markup. CDATA blocks can be very useful as long as you keep a few CDATA rules in mind.

You can use CDATA blocks when you want to include large blocks of special characters as character data. This saves the time it would take to replace characters with their corresponding entity references. However, keep in mind the size of the CDATA sections. If you serve XML files through a web service, you must ensure that client applications can handle potentially large data transfers without timing out or blocking their user interfaces. You also should make sure your server can accept large upstream transfers from clients sending XML data. Sending large blocks of data from the browser in this manner can be error-prone and tends to lock up valuable resources on the server and the client. You also should keep in mind that many users may be using mobile platforms, and there may be implications with attempting such large data transfers.

You cannot expect to keep markup untouched just because it looks as though it would be securely concealed inside a CDATA section. New programmers often assume that they can hide JavaScript or HTML markup by putting it inside a CDATA section. The content is still there, but it's treated as text.

You cannot use XML comments in a CDATA section. The literal text of the comment tags and comment text, such as `<!-- December special events -->`, is passed directly to the application or screen.

You cannot nest a CDATA section inside another CDATA section. While processing the XML document, the nested section end marker `]]>` is encountered before the real section end, which can cause a parser to assume that the nested section marker is the end of your CDATA section. This might cause a parser error when it subsequently hits the real section end.

CDATA has its own special rules. If you master them, CDATA can be a very powerful tool for your XML documents.

To see how Patricia's document is displayed with all of the added content, you'll open it in your web browser.

#### To view the sjbpet.xml file in your browser:

- 1. In your web browser, refresh or reload the **sjbpet.xml** document.
- 2. Locate the `message` element. Because different browsers display XML content differently, in the CDATA section, the CDATA tags and white space might or might not be shown in a given browser. Figure 1-33 shows the contents of the file as it appears in Firefox, Chrome, and Internet Explorer. Safari and Opera display CDATA sections in a way similar to Chrome.

Figure 1-33 Displaying the sjbpet.xml document in Firefox, Chrome, and Internet Explorer

**Firefox**

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```

- <!--
  This document contains data on SJB Pet Boutique
  holiday specials

  Filename: sjbpet.xml
  Author: Patricia Dean
  Date: 9/18/2017
-->
- <products>
- <message>
  At SJB Pet Boutique we are constantly striving to give our customers (and their humans) the highest quality products at very competitive prices. Welcome in the holidays' festivities by coming to SJB Pet Boutique for your holiday accessories. "Something for you & something for your human." Don't forget to get your SJBTracks stamps for even more discount$!!
</message>
- <product>

```

the XML declaration is hidden

CDATA tags are not shown and white space is not displayed

**Chrome**

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```

▼ <!--
  This document contains data on SJB Pet Boutique
  holiday specials

  Filename: sjbpet.xml
  Author: Patricia Dean
  Date: 9/18/2017
-->
▼ <products>
  ▼ <message>
    ▼ <![CDATA[
      At SJB Pet Boutique we are constantly striving to give our customers (and their humans) the highest quality products
      at very competitive prices. Welcome in the holidays' festivities by coming to SJB Pet Boutique for your holiday
      accessories. "Something for you & something for your human." Don't forget to get your SJBTracks stamps for even more
      discount$!!
    ]]>
  </message>
  ▼ <product>

```

the XML declaration is hidden

the entire XML declaration is displayed

CDATA tags are shown and white space is not displayed

**Internet Explorer**

```

<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<!-- This document contains data on SJB Pet Boutique holiday specials Filename: sjbpet.xml Author: Patricia Dean Date: 9/18/2017 -->
- <products>
  - <message>
    - <![CDATA[ At SJB Pet Boutique we are constantly striving to give our customers (and their humans) the highest quality products
      at very competitive prices. Welcome in the holidays' festivities by coming to SJB Pet Boutique for your holiday
      accessories. "Something for you & something for your human." Don't forget to get your SJBTracks stamps for even more discount$!! ]]>
  </message>
  - <product>
    <productName>Dog Shirt Gift Basket</productName>
    <manufacturer>SJB Pet Boutique</manufacturer>
    <description>Something for every day of the week</description>
    <price currency="USD">$35.99</price>
    <price currency="EUR">€26.79</price>
    <productItems>1200, 1201, 1202, 1203, 1204, 1205, 1206</productItems>
  </product>
- <product>

```

raw XML is shown, but with some color coding

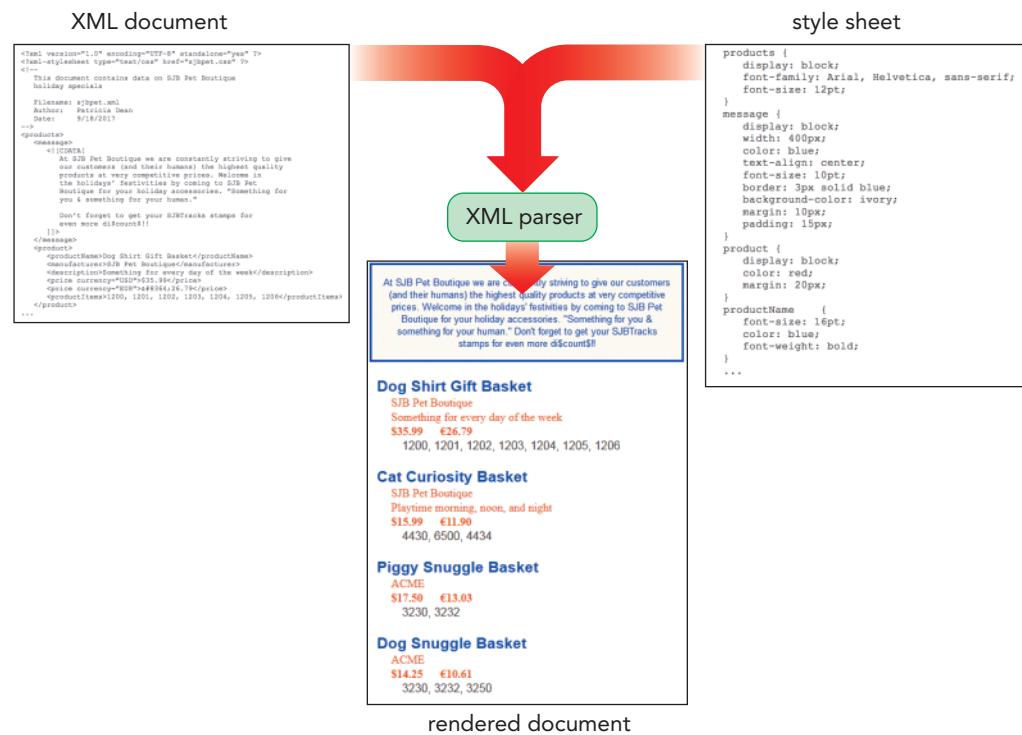
## Formatting XML Data with CSS

Patricia appreciates your work on the XML document. She would like to share this type of information with other users and place it on the web. However, she does not want to display the contents in the default hierarchical format shown in Figure 1-33. She would instead like to have the data formatted in a visually attractive way.

In contrast to HTML documents, XML documents do not include any information about how they should be rendered. Rendering is determined solely by the parser processing the document. As seen previously, different browsers render XML differently when an XML document does not indicate how its data is to be formatted or displayed. If you want to have control over the document's appearance, you have to physically link the document to a style sheet. The XML document and the style sheet are then combined by an XML parser to render a single formatted document, as shown in Figure 1-34.

Figure 1-34

Combining an XML document and a style sheet



## Applying a Style to an Element

**Cascading Style Sheets (CSS)**, the style sheet language developed for use with HTML on the web, also can be used with the elements in any XML document. CSS styles are applied to an XML element using the style declaration

```
selector {
    attribute1: value1;
    attribute2: value2;
    ...
}
```

where **selector** identifies an element (or a set of elements with each element separated by a comma) from the XML document; **attribute1**, **attribute2**, and so on are CSS style attributes; and **value1**, **value2**, and so forth are values of the CSS styles. For example, the following style declaration displays the text of the **author** element in a red boldface type:

```
author {
    color: red;
    font-weight: bold;
}
```

## REFERENCE

***Creating and Attaching a Style to an Element in an XML Document***

- To create a style rule for an element, use the syntax

```
selector {
    attribute1: value1;
    attribute2: value2;
    ...
}
```

where `selector` identifies an element (or a set of elements, with each element separated by a comma) from the XML document; `attribute1`, `attribute2`, and so on are CSS style attributes; and `value1`, `value2`, and so forth are values of the CSS styles.

- To attach a CSS style sheet to an XML document, insert the processing instruction

```
<?xml-stylesheet type="text/css" href="url" media="type" ?>
```

within the XML document's prolog, where `url` is the name and location of the CSS file, and the value of the optional `media` attribute describes the type of output device to which the style sheet should be applied. If no `media` value is specified, a default value of `all` is used.

Patricia already has generated a style sheet for the elements of the sjbpet.xml file and stored the styles in an external style sheet named sjbpet.css. To see how the styles in this style sheet will affect the appearance of the sjbpet.xml document, you must link the XML document to the style sheet.

## Inserting a Processing Instruction

You create a link from an XML document to a style sheet by using a processing instruction. A processing instruction is a command that tells an XML parser how to process the document. Processing instructions have the general form

```
<?target instruction ?>
```

where `target` identifies the program (or object) to which the processing instruction is directed and `instruction` is information that the document passes on to the parser for processing. Usually the instruction takes the form of attributes and attribute values. For example, the basic processing instruction to link the contents of an XML document to a style sheet is

```
<?xml-stylesheet type="style" href="url" media="type" ?>
```

where `style` is the type of style sheet the XML processor will be accessing, `url` is the name and location of the style sheet, and `type` is the type of output device to which the style sheet is to be applied. In this example, `xml-stylesheet` is the processing instruction's target, and the other items within the tag are processing instructions that identify the type, location, and output media for the style sheet. For a style sheet, the value of the `type` attribute should be `text/css`. The most commonly used `media` types are `screen` and `print`. If no `media` value is specified, a default value of `all` is used. The following example applies a style sheet called main.css to all output devices:

```
<?xml-stylesheet type="text/css" href="main.css" media="all" ?>
```

The same processing instruction for all output devices could be coded as follows:

```
<?xml-stylesheet type="text/css" href="main.css" ?>
```

Multiple processing instructions can exist within the same XML document for different media types. The following example shows two processing instructions being included within the same document:

```
<?xml-stylesheet type="text/css" href="main.css" media="screen" ?>
<?xml-stylesheet type="text/css" href="myPrint.css"
    media="print" ?>
```

If you prefer to avoid using the `media` attribute, the `@media` and `@import` rules for CSS can be used instead. The following example uses the `screen` media type and the `print` media type within the same CSS file:

```
@media screen {
    product {
        font-size: 12pt;
    }
}
@media print {
    product {
        font-size: 10pt;
    }
}
```

The above CSS applies a 12-point font size to the `product` element if the information is displayed on a screen, but applies a 10-point font size to the `product` element if the information is sent to a printer.

You'll add a processing instruction to the `sjbpet.xml` file to access the styles in the `sjbpet.css` file.

### To link the `sjbpet.xml` file to the `sjbpet.css` style sheet:

- 1. Return to the `sjbpet.xml` document in your text editor.
- 2. Below the XML declaration in the prolog, insert the following processing instruction, making sure not to include a space after the first question mark, and also making sure to include a space before the final question mark:

```
<?xml-stylesheet type="text/css" href="sjbpet.css" ?>
```

Figure 1-35 shows the link to the style sheet in the document.

**Figure 1-35** Inserting the processing instruction

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<?xml-stylesheet type="text/css" href="sjbpet.css" ?>
<!--
    This document contains data on SJB Pet Boutique
    holiday specials

    Filename: sjbpet.xml
    Author: Patricia Dean
    Date: 9/18/2017
-->
```

- 3. Save the changes to your document.
- 4. Refresh or reload the `sjbpet.xml` document in your web browser. Figure 1-36 shows the `sjbpet.xml` file in a browser with the `sjbpet.css` style sheet applied to the file's contents. The browser uses the specified style sheet in place of its default styles.

Figure 1-36

## The sjbpet.xml document with style sheet applied

At SJB Pet Boutique we are constantly striving to give our customers (and their humans) the highest quality products at very competitive prices. Welcome in the holidays' festivities by coming to SJB Pet Boutique for your holiday accessories. "Something for you & something for your human." Don't forget to get your SJBTracks stamps for even more di\$count\$!!

**Dog Shirt Gift Basket**  
SJB Pet Boutique  
Something for every day of the week  
**\$35.99 €26.79**  
1200, 1201, 1202, 1203, 1204, 1205, 1206

**Cat Curiosity Basket**  
SJB Pet Boutique  
Playtime morning, noon, and night  
**\$15.99 €11.90**  
4430, 6500, 4434

**Piggy Snuggle Basket**  
ACME  
**\$17.50 €13.03**  
3230, 3232

**Dog Snuggle Basket**  
ACME  
**\$14.25 €10.61**  
3230, 3232, 3250

**Trouble?** If some content is formatted differently, it is most likely due to minor rendering differences among browsers and will not cause a problem.

## INSIGHT

## Creating Style Sheets for XML with XSL

CSS is only one way of applying a style to the contents of an XML document. Another way is to use **Extensible Stylesheet Language (XSL)**, which is a style sheet language developed specifically for XML. XSL is actually an XML vocabulary, so any XSL style sheet must follow the rules of well-formed XML. XSL works by transforming the contents of an XML document into another document format. For example, an XSL style sheet can be used to transform the contents of an XML document into an HTML file that can be displayed in any web browser. In addition, the HTML code generated by an XSL style sheet could itself be linked to a CSS file.

XSL is not limited to generating HTML code; an XML document instead could be transformed into another XML document.

Web browsers often use internal XSL style sheets to display XML documents. The outline form of Patricia's document shown in Figure 1-33 is actually the XML document as transformed by the XSL style sheet built into each browser's XML parser. This style sheet is used by the browser unless a different style sheet is specified by a processing instruction in the XML document.

You've finished creating the XML vocabulary for the holiday specials products at SJB Pet Boutique. Patricia anticipates creating future documents that combine elements from this vocabulary with elements from other XML vocabularies. She'd like you to ensure that you'll be able to distinguish elements in one vocabulary from elements in another vocabulary. XML enables you to do this using namespaces.

## Working with Namespaces

A namespace is a defined collection of element and attribute names. For example, the collection of element and attribute names from Patricia's products vocabulary could define a single namespace. Applying a namespace to an XML document involves two steps:

1. Declare the namespace.
2. Identify the elements and attributes within the document that belong to that namespace.

### Declaring a Namespace

To declare a namespace for an element within an XML document, you add an attribute within the opening tag for the element using the syntax

```
<element xmlns:prefix="uri"> ... </element>
```

where *element* is the element in which the namespace is declared, *prefix* is a string of characters that you'll add to element and attribute names to associate them with the declared namespace, and *uri* is a Uniform Resource Identifier (URI)—a text string that uniquely identifies a resource. In this case, the URI is the declared namespace. For example, the following code declares a namespace with the URI <http://example.com/sjbpetboutique/products> and associates that URI with the prefix *prd* within the *products* element:

```
<products xmlns:prd="http://example.com/sjbpetboutique/products">
...
</products>
```

The number of namespace attributes that can be declared within an element is unlimited. In addition, a namespace that has been declared within an element can be applied to any descendant of the element. Some XML authors add all namespace declarations to a document's root element so that each namespace is available to all elements within the document.

### Applying a Default Namespace

You can declare a **default namespace** by omitting the prefix in the namespace declaration. Any descendant element or attribute is then considered part of this namespace unless a different namespace is declared within one of the child elements. The syntax to create a default namespace is

```
<element xmlns="uri"> ... </element>
```

For instance, to define the <http://example.com/sjbpetboutique/products> namespace as the default namespace for all elements in the document, you could use the following root element:

```
<products xmlns="http://example.com/sjbpetboutique/products">
```

In this case, all elements in the document, including the *products* element, are considered part of the <http://example.com/sjbpetboutique/products> namespace.

**REFERENCE**

### Declaring a Namespace

- To declare a namespace for an element within an XML document, add the `xmlns:prefix` attribute to the opening tag of the element using the syntax  
`<element xmlns:prefix="uri"> ... </element>`  
where `element` is the element in which the namespace is declared, `prefix` is the namespace prefix, and `uri` is the URI of the namespace.
- To declare a default namespace, add the `xmlns` attribute without specifying a prefix, as follows:  
`<element xmlns="uri"> ... </element>`

The advantage of default namespaces is that they make the code easier to read and write because you do not have to add the namespace prefix to each element. The disadvantage, however, is that an element's namespace is not readily apparent from the code. Still, many compound documents use a default namespace that covers most of the elements in the document, and assign namespace prefixes to elements from other XML vocabularies. A **compound document** is an XML document composed of elements from other vocabularies or schemas. For example, you may combine elements from HTML and XML vocabularies. You'll learn how to work with compound documents in future tutorials.

**INSIGHT**

### Understanding URLs

The URI used in namespaces looks like a web address used to create a link to a website; however, that is not its purpose. The purpose of a URL is simply to provide a unique string of characters that identify a resource.

One version of a URI is the Uniform Resource Locator (URL), which is used to identify the location of a resource (such as a web page) on the web. There is a good reason to also use URLs as a basis for identifying namespaces. If an XML vocabulary is made widely available, the namespace associated with that vocabulary must be unique. URLs serve as a built-in mechanism on the web for generating unique addresses. For example, assume that the home page of Patricia's company, SJB Pet Boutique, has the web address

`http://example.com/sjbpetboutique`

This address provides customers with a unique location to access all of SJB Pet Boutique's online products. To ensure the uniqueness of any namespaces associated with the vocabularies developed for SJB Pet Boutique documents, it makes sense to use the SJB Pet Boutique web address as a foundation. Note that although a URI doesn't actually need to point to a real site on the web, it's often helpful to place documentation at the site identified by a URI so users can go there to learn more about the XML vocabulary being referenced.

The use of URLs is widely accepted in declaring namespaces, but you can use almost any unique string identifier, such as SJBPetBoutiqueProductNS or P2205X300x. The main requirement is that a URI is unique so that it is not confused with the URLs of other namespaces.

Patricia wants you to declare a default namespace for the products vocabulary. You'll declare the namespace in the root element of the sjbp.xml file. The URI for the namespace is `http://example.com/sjbpetboutique/products`. This URI does not point to an actual site on the web, but it does provide a unique URI for the namespace.

### To declare a default namespace:

- 1. Return to the **sjbpet.xml** file in your text editor.
- 2. Within the opening `<products>` tag, insert the following default namespace declaration:

```
xmlns="http://example.com/sjbpetboutique/products"
```

Figure 1-37 shows the default namespace declaration in the document.

Figure 1-37

The sjbpet.xml document with default namespace applied

A screenshot of an XML document titled "sjbpet.xml". The document contains the following code:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<?xml-stylesheet type="text/css" href="sjbpet.css" ?>
<!--
This document contains data on SJB Pet Boutique
holiday specials

Filename: sjbpet.xml
Author: Patricia Dean
Date: 9/18/2017
-->
<products xmlns="http://example.com/sjbpetboutique/products">
    <message>
```

A green callout box points to the line `<products xmlns="http://example.com/sjbpetboutique/products">` with the text "default namespace added to the root element". A red arrow points to the same line from the text "Figure 1-37 shows the default namespace declaration in the document."

- 3. Press **Ctrl+S** to save your work.
- 4. Reload or refresh **sjbpet.xml** in your web browser. If you're using any major browser, the web page should look unchanged from Figure 1-36.

**Trouble?** If your browser reports a syntax error, return to the sjbpet.xml file in your text editor, ensure that the namespace is within the opening tag of the root element, save any changes, and then repeat Step 4.

Patricia is very happy with the work you've done. She'll show your work to the other members of her web team, and if they need more XML documents created in the future, they'll get back to you.

### REVIEW

#### Session 1.2 Quick Check

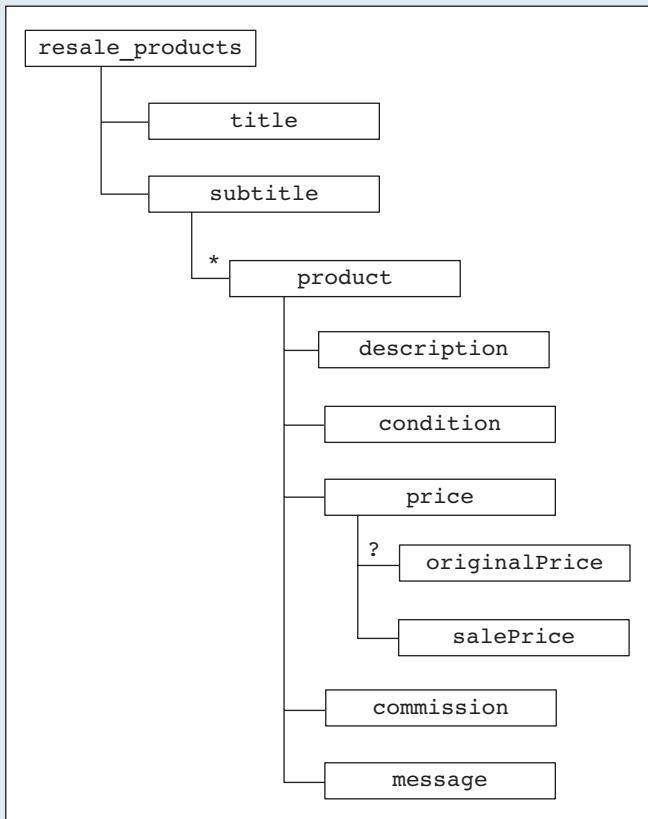
1. What is the error in the following code?  
`<Title>Grilled Steak</title>`
2. What is the root element?
3. What are sibling elements?
4. What is parsed character data?
5. Name three ways to insert the ampersand ( & ) symbol into the contents of an XML document without it being treated as parsed character data.
6. What is a CDATA section?
7. What is a processing instruction?
8. What code links an XML document to a style sheet file named standard.css?
9. What is a namespace?
10. What attribute would you add to a document's root element to declare a default namespace with the URI `http://ns.doc.book`?

## Review Assignments

**Data Files needed for the Review Assignments: resale.css, resale.txt**

Patricia would like you to create another document for the SJB Pet Boutique resale business. She has provided you with a text file that she'd like you to convert to XML. Figure 1-38 outlines the structure of the final document that she wants you to create.

Figure 1-38 The SJB Pet Boutique resale tree hierarchy



Patricia also would like the final document displayed using the provided *resale.css* style sheet, as shown in Figure 1-39.

**Figure 1-39** The resale.xml document in the browser

**Resale Products**  
*Latest Offerings*

**8" Hamster Travel Cage**

Good

- \$35
- \$15
- 15%

**Dog and Cat Doorstop**

Excellent

- \$70
- \$20
- 10%

**Puppy Dog Rhinestone Collar**

Excellent

- \$25
- \$10
- 15%

**30" Kitty Travel Cage**

New

- \$40
- \$35
- 20%

Interested in resale? The SJB Pet Boutique will publish for resale select items purchased at our store that you or your owner have outgrown. All sales are "as-is" condition. A commission is charged for listing the item for resale. If your resale item is sold and you purchase a new item at our store using the money from the sale of the item, you can save 50% off our commission price! ☺

Complete the following:

1. Using your text editor, open the **resale.txt** file located in the **xml01 ▶ review** folder included with your Data Files.
2. Save the document as **resale.xml**.
3. Create a prolog at the top of the document indicating that this is an XML document using the UTF-8 encoding scheme, and that it is a standalone document.
4. Create processing instructions to link the **resale.xml** document to the **resale.css** style sheet.
5. On a new line below the XML declaration, insert a comment containing the text

**SJB Pet Boutique resale products, latest offerings**

**Filename: resale.xml**

**Author: your name**

**Date: today's date**

where *your name* is your first and last names, and *today's date* is the current date.

6. Enclose the document body content in a root element named **resale\_products**.
7. Mark the text *Resale Products* with an element named **title**.
8. Mark the text *Latest Offerings* with an element named **subtitle**.
9. There are four new resale products. Create an element named **product**, which will contain all of the information about each product within a single product element.
10. Each **product** element should now contain five detail lines about each product, as shown in Figure 1-39.
11. Mark the first line of each product with an element named **description**, which contains a description of the product.

12. Mark the second line of each product with an element named **condition**, which contains data regarding the condition of the product.
13. Mark the third line of each product with an element named **originalPrice**, which will contain data regarding the original price of the product.
14. Mark the fourth line of each product with an element named **salePrice**, which will contain data regarding the resale price of the product.
15. Mark the fifth line of each product with an element named **commission**, which will contain data regarding the commission percentage to be earned on the sale.
16. Create a parent element named **price** for the child elements **originalPrice** and **salePrice**.
17. At the bottom of the document is a message to clients who are interested in resale. Enclose this message in a CDATA section and place the CDATA section within an element named **message**.
18. Patricia wants a smiley face ( ☺ ) to be displayed at the end of the contents of the **message** element. Delete the colon and closing parenthesis ( : ) ) at the end of the message text. Outside the CDATA section but within the **message** element, add the character reference `&#9786;`. (Note: The reference must be outside the CDATA section in order for parsers to translate it.)
19. Within the opening **<resale\_products>** tag, insert a default namespace declaration to place all elements in the **http://example.com/sjbpetboutique/resale** namespace.
20. Save your changes to the **resale.xml** document.
21. Open the **resale.xml** document in your web browser. Compare the results to Figure 1-39. If your output does not match the expected output, correct any errors and refresh your browser to verify changes.

APPLY

## Case Problem 1

**Data Files needed for this Case Problem:** **membership.css**, **membership.txt**

**MSN Baseball Fan Club** The MSN Baseball Fan Club is an online fan club based in Madison, Wisconsin. Anyone can be a member of the club; the only prerequisite is that the person is a fan of Madison baseball. Members have voted to add to the current website a page to display each club member's blog entries. The club wants the output displayed with a style sheet, which one of the members has provided. Figure 1-40 outlines the structure of the document you will create.

**Figure 1-40** Baseball membership tree hierarchy of combined documents

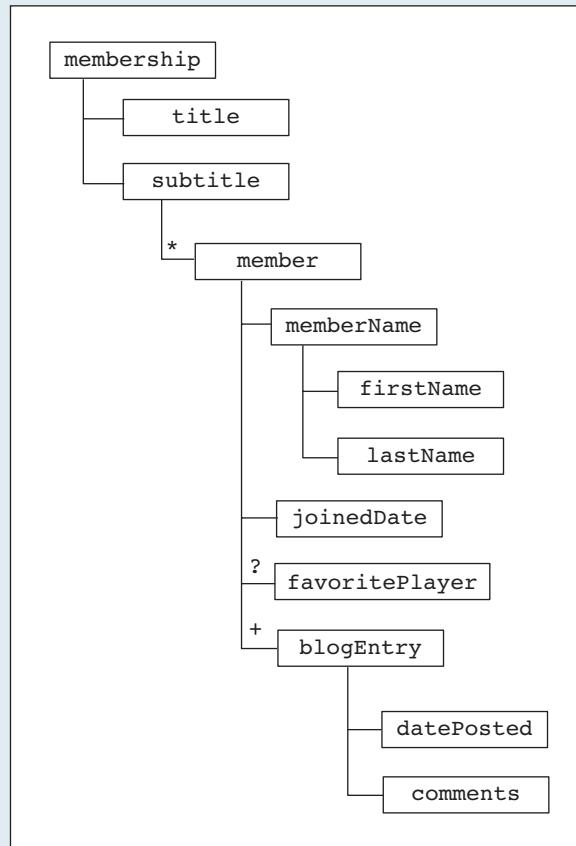


Figure 1-41 shows the document displayed with a style sheet.

**Figure 1-41** The membership.xml document displayed using the membership.css style sheet

**MSN Baseball Fan Club**  
*Blog Entries for Members*

---

**Johnny BeGood**  
1962/2/1  
**Ryan Braun**

2011/10/10  
Did you see the game on 10/7? Braun is crazy awesome!!

2010/3/10  
I knew when my career was over. In 1965 my baseball card came out with no picture.

---

**Bob Uecker**  
2001/8/21  
**J Scanlon**

2003/7/1  
The .247 avg doesn't do him justice. He is more like a .300 player, but I think maybe more game time is needed. For playing IF, he does a great job. Really quick on the feet. Hope he returns next year. Go Mallards!

---

**Tom Churchill**  
2010/4/21  
**Willie Mays**

2011/12/19  
I know Willie Mays isn't playing anymore, but as he said 'I think I was the best baseball player I ever saw'. Enough Said.

2010/4/21  
I just found this blog. I probably won't be posting much as I am too busy going to the games to post about them. Seriously though... I cannot wait for next season. My team is already out of the running, so I wait.

Complete the following:

1. Using your text editor, open the **membership.txt** file located in the **xml01 ▶ case1** folder included with your Data Files.
2. Save the document as **membership.xml**.
3. Create a prolog at the top of the *membership.xml* document indicating that this is an XML document using the UTF-8 encoding scheme, and that it is a standalone document.
4. Create a processing instruction to link the *membership.xml* document to the **membership.css** style sheet.
5. Directly below the XML declaration, insert a comment containing the text  
**MSN Baseball Fan Club Blog Entries**

**Filename: membership.xml**

**Author:** *your name*

**Date:** *today's date*

where *your name* is your first and last names, and *today's date* is the current date.

6. Enclose the document body content in a root element named **membership**.
7. Enclose the text *MSN Baseball Fan Club* in an element named **title**.
8. Enclose the text *Blog Entries for Members* in an element named **subtitle**.
9. Three members have had data collected for the document. Enclose the information about each member within an element named **member**.  
Each **member** element should contain four detail lines about the member, as shown in Figure 1-41, prior to adding the blog content. Be sure the **member** element encloses all information about a member, including his blog entry(ies).
10. Mark the first line of each **member** element, which contains the member's first name, with an element named **firstName**.
11. Mark the second line of each **member** element, which contains the member's last name, with an element named **lastName**.
12. For each member, create a parent element named **memberName** that contains **firstName** and **lastName** as child elements.
13. Mark the third line of each member's information, which contains the date the member joined the fan club, with an element named **joinedDate**.
14. Mark the fourth line of each member's information, which contains the name of the member's favorite player, with an element named **favoritePlayer**.
15. Each member has at least one blog entry, which contains a date and a comment as shown in Figure 1-41. Enclose each date and comment within a **blogEntry** element. You may need to add more than one **blogEntry** element for a member who has multiple blog entries.
16. Enclose the date associated with each blog entry in a **datePosted** element.
17. Enclose the contents of each **comments** element in a CDATA section, with the **<comments>** tags outside of the CDATA section.
18. Insert a namespace declaration to add all the elements in the document body to the **http://example.com/msnmembball/membership** namespace.
19. Save your changes to the *membership.xml* document.
20. Open the **membership.xml** document in your web browser and compare the output to Figure 1-41. If necessary, correct any errors and re-verify the output in the browser.

**APPLY****Case Problem 2****Data Files needed for this Case Problem: chester.css, chester.txt**

**Chester's Restaurant** Chester's Restaurant is located in the quaint town of Hartland, Minnesota. Jasmine Pup, the owner and operator, wants to display the recently revamped breakfast menu on the web. She wants you to apply an existing style sheet to the document so the menu is nicely formatted on the web. She has saved the menu information to a text file and needs you to convert the document to XML. Figure 1-42 outlines the structure of the document you will create.

**Figure 1-42** Chester's breakfast menu tree hierarchy

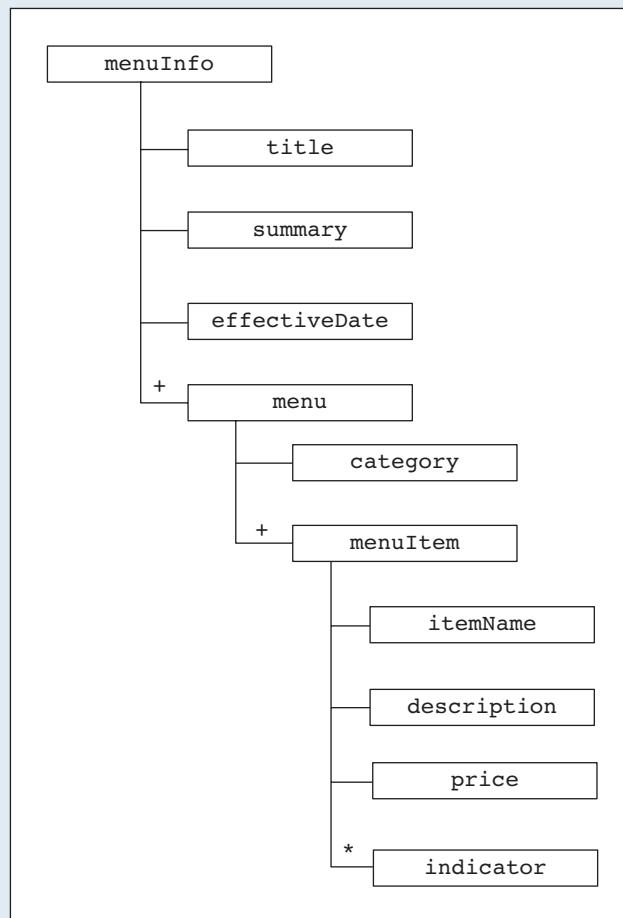


Figure 1-43 shows the document displayed with a style sheet.

Figure 1-43 The chester.xml document displayed using the chester.css style sheet

# Chester's Breakfast Menu

If you've been craving an authentic homestyle country breakfast, look no further than Chester's! We've got your breakfast favorites served up just the way you like them!!

03/12/2016

## Traditional Favorites

- **Rise n' Shine**  
Two Eggs\* cooked to order with Grits, Gravy and Homemade Buttermilk Biscuits along with real Butter and the best fresh jam available. Served with your choice of Fresh Fruit or Hashbrown Casserole and Smoked Sausage Patties, Turkey Sausage Patties or Thick-Sliced Bacon.  
**7.95**
- **Fresh Mornin' Sampler**  
Low-Fat Vanilla Yogurt and Seasonal Fruit topped with our Honey Granola mix of Almonds and Dried Fruit. Served with a Wild Maine Blueberry Muffin or an Apple Bran Muffin.  
**6.95** ♥ ♦ ♣

## Lite and Quick

- **Oatmeal Breakfast**  
Our Oatmeal is served warm with your choice of Fried Apples, Pecans, Raisins, Fresh Sliced Bananas or 100% Pure Natural Syrup. Also, served with your choice of Apple Bran Muffin or Wild Maine Blueberry Muffin. Available all day.  
**6.95** ♥ ♦ ♣
- **Chester's Meat Platter**  
Country Ham, Pork Chops or Steak\* grilled to order, Three Eggs\* cooked to order served with Cottage Cheese, Smoked Sausage Patties, Turkey Sausage Patties or Thick-Sliced Bacon.  
**12.95** └─

Complete the following:

1. Using your text editor, open the **chester.txt** file located in the **xml01 ▶ case2** folder included with your Data Files.
2. Save the document as **chester.xml**.
3. Create a prolog at the top of the document indicating that this is an XML document using the UTF-8 encoding scheme, and that it is a standalone document.
4. Create a processing instruction to link the **chester.xml** document to the **chester.css** style sheet.
5. Directly below the XML declaration, insert a comment containing the text

### **Chester's Restaurant Breakfast Menu**

**Filename:** **chester.xml**

**Author:** *your name*

**Date:** *today's date*

where *your name* is your first and last names, and *today's date* is the current date.

6. Enclose the document body content in a root element named **menuInfo**.
7. Enclose the text *Chester's Breakfast Menu* in an element named **title**.
8. Enclose the three lines of text beginning with *If you've been craving* in an element named **summary**. Enclose the content for each **summary** element in a CDATA section.
9. Enclose the date found after the text *Effective thru* in an element named **effectiveDate**.
10. Add an attribute called **text** to the **effectiveDate** element with a value of **Effective thru**.

11. The menu information is divided into two categories. Add the first opening tag for the parent element called **menu** to the document on the line before the text *Traditional Favorites*. Add the closing tag for the first **menu** element and the opening tag for the second **menu** element on the line before the text *Lite and Quick*. Add the closing tag for the second **menu** element on the last line of the document.
12. Each menu category contains a description to be marked with a **category** element. Mark the text *Traditional Favorites* in the first **menu** element with a **category** element, and then mark the text *Lite and Quick* in the second **menu** element with another **category** element.
13. Each menu category also contains one or more child **menu** items and their associated information. On the next line below each **category** element, add a parent element named **menuItem**. Create element tags for each of the following child elements and place them within each **menuItem** parent element:
  - a. Mark the first line of each menu category with an element named **itemName**, which will contain the name of the item on the menu. Be sure to place this element within the **menuItem** parent element.
  - b. Mark the second piece of information for each menu category with an element named **description**, which will give a detailed description of what the menu item consists of. There may be several lines for the description. The content of the **description** element should be enclosed within a CDATA section. Be sure to place this element within the **menuItem** parent element.
  - c. Mark the third piece of information for each menu category with an element named **price**, which will indicate the purchase price of the item. Be sure to place this element within the **menuItem** parent element.
  - d. Mark the fourth piece of information for each menu category with an element named **indicator**, which contains optional text indicators that flag the menu item under one or more of the following subcategories:
    - ♥ Heart Healthy: character reference &#9829;
    - ♦ Low Sodium: character reference &#9830;
    - ♠ Vegan: character reference &#9824;
    - ♣ Low Carb: character reference &#9832;Because each menu item could fall under none or several categories, the number of indicator elements will vary. If a menu item includes an **indicator** element, use the appropriate character reference to display the symbol for the indicator instead of displaying the text description. Be sure to place this element within the **menuItem** parent element.
  - e. Because others might not be familiar with the character references, include a single-line comment containing the text description for each character reference after the closing tag for each **indicator** element.
  - f. If there are more **menuItem** elements, repeat Steps 13a–13e.
14. Add the namespace <http://example.com/chesterhartland/menu> to the opening tag of the root element.
15. Save your changes to the **chester.xml** document.
16. Open the **chester.xml** document in your web browser and compare the output to Figure 1-43. If necessary, correct any errors and then re-verify the output in the browser.

## Case Problem 3

**Data Files needed for this Case Problem:** `recipe.txt`, `weekendFun.css`, `weekendFuntxt.xml`

**Weekend Fun Snacks** Cleo Coal is creating a website called *Weekend Fun Snacks*. The site will list her picks of the best and easiest recipes for kids to cook on the weekend (or anytime). Cleo has been entering recipe information into an XML document. However, she has run into some problems and has come to you for help. Cleo would like your help with cleaning up her XML document so it displays in a browser with her style sheet. Once you've corrected the XML document, she also would like you to add a new recipe to the document. Figure 1-44 displays a preview of the document's contents when all corrections are made and the new recipe is added.

Figure 1-44 The corrected `weekendFun.xml` document

The screenshot shows a web page titled "Weekend Fun Snacks". It features three recipe cards with titles, descriptions, ingredients, and preparation steps.

- Fruit Saber**: Description: "Need an idea for a fun weekend snack? Try one of our simple recipes!"  
Ingredients: 1 pear, 1 banana, 1 strawberry, 1 orange, 4 wooden skewers.  
Preparation: Cut all fruit in large chunks; Skewer the chunks, alternating between the different types of fruit.
- Peanut Butter Drops**: Description: "Great for peanut butter lovers!"  
Ingredients: ½ cup honey, ½ cup peanut butter, 1 cup nonfat dry milk, 1 cup Rice Krispies cereal, wax paper.  
Preparation: Mix all ingredients and then roll the mixture into 1" balls; Lay the balls on waxed paper; Refrigerate until set.
- Leaf Piles**: Description: "Jump into a corn flake leaf pile. Nice snack for fall."  
Ingredients: 6 cups corn flakes, 1 cup Karo syrup, 1 cup peanut butter, wax paper.  
Preparation: In a microwave, melt the Karo and peanut butter together; Pour over the corn flakes and place them in 2" piles on wax paper; Let them cool and dry.

Complete the following:

1. Using your text editor, open the `weekendFuntxt.xml` file located in the `xml01 ▶ case3` folder included with your Data Files, and then save the document as `weekendFun.xml`.
2. Use <http://validator.w3.org> to identify the first error in the code.

-  **EXPLORE** 3. Using the error information reported, locate and correct the first error.
4. Save the modifications to **weekendFun.xml** and then check it again for well-formedness.
  5. If the parser reports another error, repeat Steps 3 and 4 until all errors are corrected. The following are some errors to look for:
    - misspelled element names
    - missing quotes
    - misplaced closing tags
    - missing closing tags
    - invalid namespace
  6. Save the modifications to **weekendFun.xml**.
  7. Open the document in your browser.
  8. Verify that the style sheet has been applied. If not, edit and reload the document in the web browser.
-  **EXPLORE** 9. Modify the prolog at the top of the document to indicate that the document uses the UTF-8 encoding scheme, and that it is a standalone document.
-  **EXPLORE** 10. Copy and paste the content from the **recipe.txt** document into the **weekendFun.xml** document as a new recipe. Using the existing recipe vocabulary as your guide, modify the newly added recipe data to use the vocabulary for the data.
11. Save the modifications to **weekendFun.xml**.
  12. Reload the document in the browser.
  13. Verify that all of the information for the new recipe is included, as shown in Figure 1-44.
-  **EXPLORE** 14. Draw the tree structure for the contents of the **weekendFun.xml** document. Each document can have many recipes, an optional description, at least one ingredient, an optional measurement, and at least one direction. Each other element occurs one time.

**CREATE**

## Case Problem 4

**Data Files needed for this Case Problem:** **rycheBooks.css**, **rycheBooks.txt**, **rycheBooksPrint.css**

**Ryche Books** David Ryche is the owner and operator of Ryche Books, which is an online bookstore that specializes in hard-to-find books. He often receives phone calls regarding his current stock of books and he would like this information available to his customers at his website. He has created a text file that contains data about his current inventory of books. He would like your help with converting his text file to an XML document and then displaying that information in a web browser. Figure 1-45 gives a preview of the document's contents in a browser.

**Figure 1-45** The browser output for the rycheBooks.xml document

<p><b>The Titan's Curse</b></p> <p>Rick Riordan      Softcover      Penguin Books      <b>7.95</b></p> <ul style="list-style-type: none"> <li>• children</li> <li>• teens</li> <li>• top-seller</li> <li>• fiction</li> <li>• adventure</li> </ul>
<p><b>The Maze of Bones</b></p> <p>Rick Riordan      Hardcover      Scholastic      <b>10.95</b></p> <ul style="list-style-type: none"> <li>• children</li> <li>• teens</li> <li>• top-seller</li> <li>• fiction</li> <li>• adventure</li> </ul>
<p><b>100 Words Kids Need to Read by 3rd Grade</b></p> <p>Scholastic      Softcover      Scholastic      <b>5.95</b></p> <ul style="list-style-type: none"> <li>• children</li> <li>• non-fiction</li> <li>• educational</li> </ul>
<p><b>Harry Potter and the Deathly Hallows (Harry Potter #7)</b></p> <p>J. K. Rowling      Hardcover      Bloomsbury Publishing      <b>12.95</b></p> <ul style="list-style-type: none"> <li>• children</li> <li>• teens</li> <li>• adult</li> <li>• top-seller</li> <li>• fiction</li> <li>• adventure</li> </ul>

Figure 1-46 shows a print preview of the document's contents.

**Figure 1-46** The print preview output for the rycheBooks.xml document

<p><b>The Titan's Curse</b></p> <table> <tr> <td>Rick</td><td>Riordan</td><td>Softcover</td><td>Penguin Books</td><td><b>7.95</b></td></tr> <tr> <td>children</td><td>teens</td><td>top-seller</td><td>fiction</td><td>adventure</td></tr> </table>	Rick	Riordan	Softcover	Penguin Books	<b>7.95</b>	children	teens	top-seller	fiction	adventure
Rick	Riordan	Softcover	Penguin Books	<b>7.95</b>						
children	teens	top-seller	fiction	adventure						
<p><b>The Maze of Bones</b></p> <table> <tr> <td>Rick</td><td>Riordan</td><td>Hardcover</td><td>Scholastic</td><td><b>10.95</b></td></tr> <tr> <td>children</td><td>teens</td><td>top-seller</td><td>fiction</td><td>adventure</td></tr> </table>	Rick	Riordan	Hardcover	Scholastic	<b>10.95</b>	children	teens	top-seller	fiction	adventure
Rick	Riordan	Hardcover	Scholastic	<b>10.95</b>						
children	teens	top-seller	fiction	adventure						
<p><b>100 Words Kids Need to Read by 3rd Grade</b></p> <table> <tr> <td>Scholastic</td><td>Softcover</td><td>Scholastic</td><td><b>5.95</b></td></tr> <tr> <td>children</td><td>non-fiction</td><td>educational</td><td></td></tr> </table>	Scholastic	Softcover	Scholastic	<b>5.95</b>	children	non-fiction	educational			
Scholastic	Softcover	Scholastic	<b>5.95</b>							
children	non-fiction	educational								
<p><b>Harry Potter and the Deathly Hallows (Harry Potter #7)</b></p> <table> <tr> <td>J. K. Rowling</td><td>Hardcover</td><td>Bloomsbury Publishing</td><td><b>12.95</b></td></tr> <tr> <td>children</td><td>teens</td><td>adult</td><td>top-seller</td><td>fiction</td><td>adventure</td></tr> </table>	J. K. Rowling	Hardcover	Bloomsbury Publishing	<b>12.95</b>	children	teens	adult	top-seller	fiction	adventure
J. K. Rowling	Hardcover	Bloomsbury Publishing	<b>12.95</b>							
children	teens	adult	top-seller	fiction	adventure					

Complete the following:

1. Using your text editor, open the **rycheBooks.txt** file, which is located in the **xml01 ▶ case4** folder included with your Data Files.
2. Save the document as **rycheBooks.xml**.
3. At the top of the document, add code indicating that this is an XML document using the UTF-8 encoding scheme, and that it is a standalone document.
4. Create processing instructions to link the XML document to the **rycheBooks.css** style sheet for screen media, and to the **rycheBooksPrint.css** style sheet for print media.
5. Within the document's prolog, include the filename, your name, today's date, and the purpose of the document in a comment.
6. Mark up the contents of the document using the following specifications:
  - The root element of the document should be named **books**.
  - The **books** element should contain multiple occurrences of a child element named **book**.
  - Each **book** element should have a single attribute named **ISBN** containing the International Standard Book Number (ISBN) of the book. (Note: The ISBN begins with the letter I, which is followed by several numeric digits.)
  - Each **book** element should have six child elements: **title**, **author**, **type**, **publisher**, **sellPrice**, and **categories**.
    - The **title** element should contain the book's title.
    - The **author** element should contain three child elements: **firstName**, **middleName**, and **lastName**. Each author name should be divided accordingly and stored in these elements. (Note: If the author is a business name such as *Scholastic*, it will not have all three child elements.)
    - The **type** element should contain the book's cover type: **hardcover** or **softcover**.
    - The **publisher** element should contain the name of the book's publisher.
    - The **sellPrice** element should contain the book's selling price.
    - The **categories** element should contain at least one child element called **category**. A book may have multiple **category** elements.
7. Add a default namespace for the books vocabulary using an appropriate URI.
8. Save your changes to the **rycheBooks.xml** document.
9. Open the **rycheBooks.xml** document in your web browser and compare the output to Figure 1-45.
10. If your web browser reports any syntax errors, locate and correct each error using the information from the web browser. Save any modifications to **rycheBooks.xml** and reload the document in your browser.
11. Verify that the style sheet is also properly applied to print media. To verify the print media style sheet, use the print preview feature of your browser and compare the output to Figure 1-46. If necessary, correct any errors and re-verify the output in the browser.



**OBJECTIVES****Session 2.1**

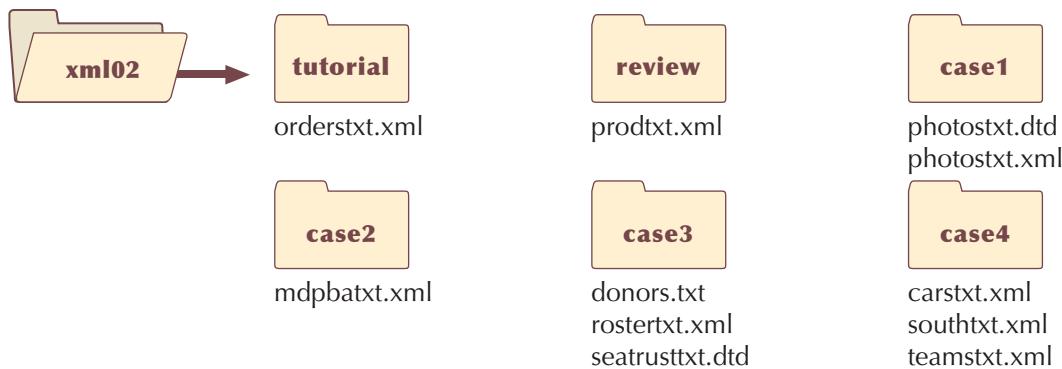
- Review the principles of data validation
- Create a DOCTYPE
- Declare XML elements and define their content
- Define the structure of child elements

**Session 2.2**

- Declare attributes
- Set rules for attribute content
- Define optional and required attributes
- Validate an XML document

**Session 2.3**

- Place internal and external content in an entity
- Create entity references
- Understand how to store code in parameter entities
- Create comments in a DTD
- Understand how to create conditional sections
- Understand how to create entities for non-character data
- Understand how to validate standard vocabularies

**STARTING DATA FILES**

# Validating Documents with DTDs

*Creating a Document Type Definition for Map Finds For You*

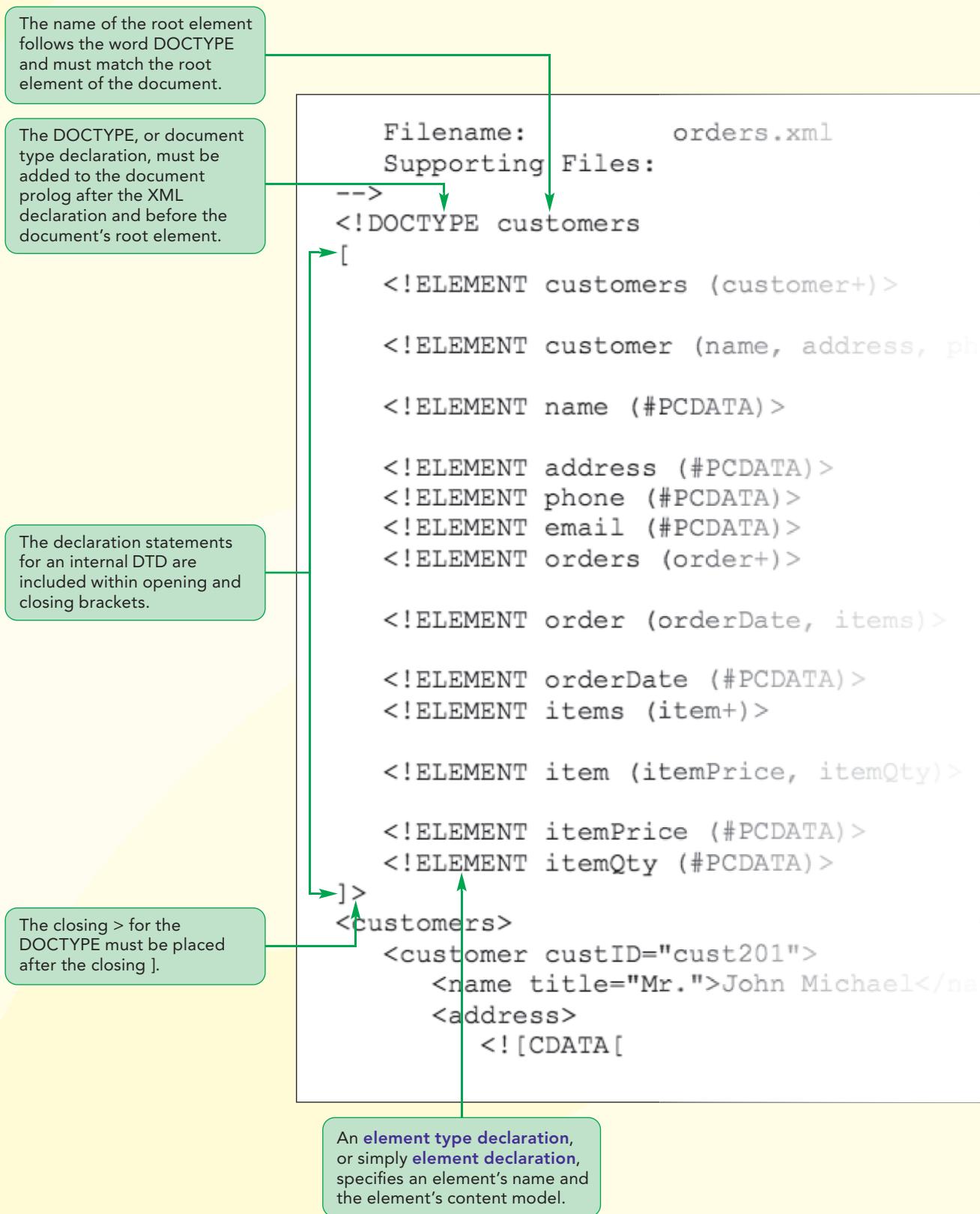
## Case | Map Finds For You

Benjamin Mapps works at Map Finds For You, an online store that sells mass-produced and custom map products. Map Finds For You sells more than 1000 map products ranging from maps of the present-day world to detailed historical maps. Some are available in both paper and electronic formats. Part of Benjamin's job at Map Finds For You is to record information about the store's customers, including the individual orders they place.

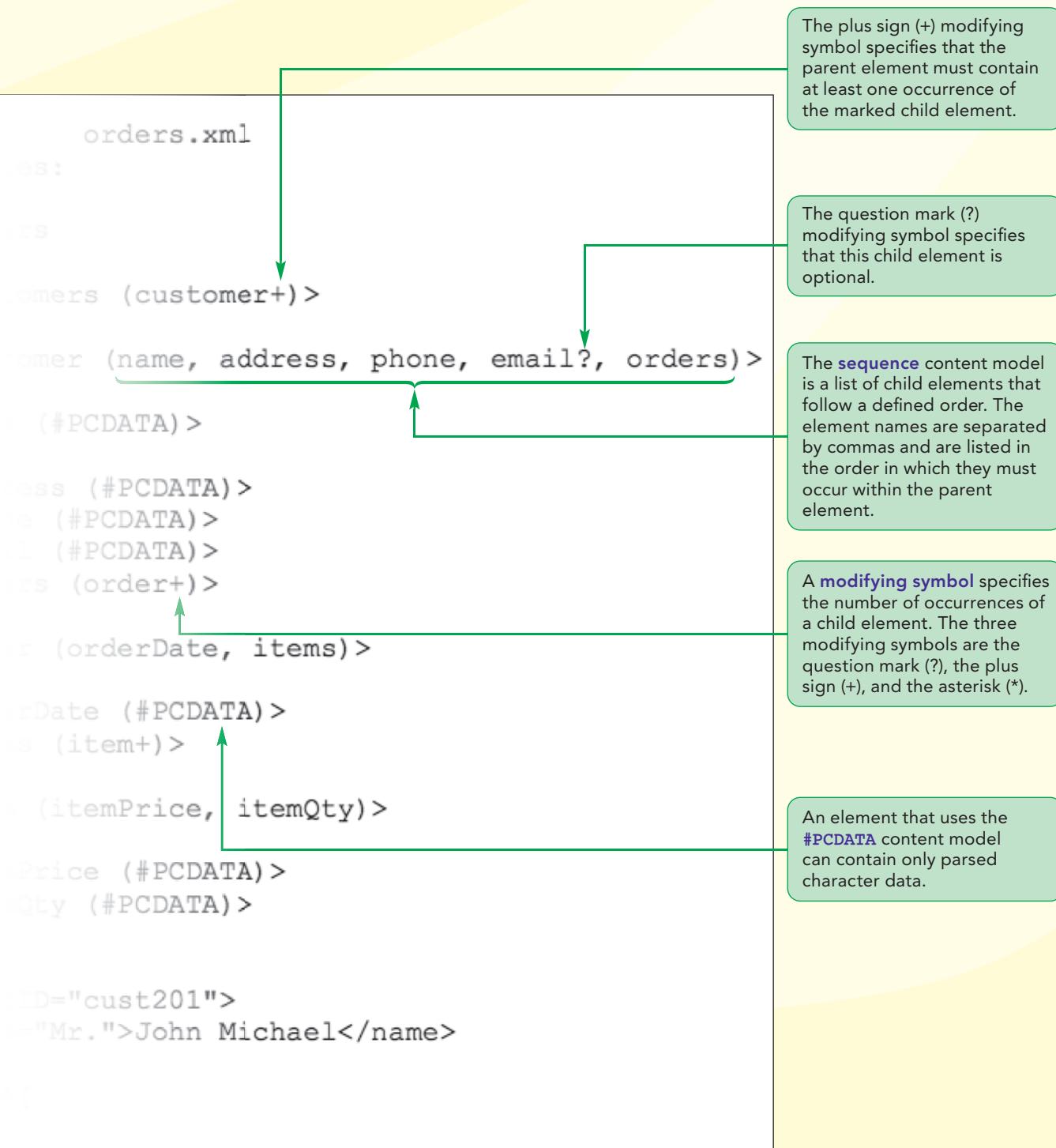
Benjamin is starting to use XML to record this information, and he has created a sample XML document containing information on customers and their orders. Benjamin knows that his document needs to be well formed, following the XML syntax rules exactly, but he also wants the document to follow certain rules regarding content. For example, data on each customer must include the customer's name, phone number, and address. Each customer order must contain a complete list of the items purchased and include the date the order was placed. You will create an XML document for Benjamin that adheres to both the rules of XML and the rules Benjamin has set up for the document's content and structure.

Note: To complete this tutorial, you need access to a recent version of a major browser, such as Internet Explorer, Firefox, or Chrome, as well as an XML parser capable of validating an XML document based on a DTD.

# Session 2.1 Visual Overview:



# The Structure of a DTD



## Creating a Valid Document

Benjamin has created a sample document that contains information about Map Finds For You's customers. To keep the information to a manageable size, Benjamin limited the document to three customers and their orders. Figure 2-1 shows a table of the information he entered for those customers.

Figure 2-1

Customer orders table

Customer		Orders		Item	Qty	Price	Sale Item
name:	Mr. John Michael	orderID:	or1089	WM100PL	1	39.95	N
custID:	cust201	orderBy:	cust201	WM101P	2	19.90	Y
address:	41 West Plankton Avenue Orlando, FL 32820	orderDate:	8/11/2017				
phone:	(407) 555-3476						
email:	jk@example.net						
name:	Mr. Dean Abernath	orderID:	or1021	WM100PL	1	29.95	N
custID:	cust202	orderBy:	cust202	WM105L	1	19.95	N
address:	200 Bear Avenue Front Royal, VA 22630	orderDate:	8/1/2017				
phone:	(540) 555-1788	orderID:	or1122	H115E	2	24.90	Y
email:	dabernath@example.com	orderBy:	cust202	H115F	1	14.95	N
name:	Riverfront High School	orderID:	or1120	WM140P	2	78.90	N
custID:	cust203	orderBy:	cust203				
address:	1950 West Magnolia Drive River Falls, WI 54022	orderDate:	9/15/2017				
phone:	(715) 555-4022						
email:							

For each customer, Benjamin has recorded the customer's name, ID, address, phone number, and email address. Each customer has placed one or more separate orders. For convenience, Benjamin has grouped each customer order within an `orders` element. For each order, Benjamin recorded the order's ID number, customer ID number, and date. Finally, for each item ordered, he entered the item number, the quantity, the price of the item, and whether the item was on sale. Benjamin placed this information in an XML document, which you will open now.

### To open the `orders` document:

- 1. Use your text editor or XML editor to open `orderstxt.xml` from the `xml02` ► tutorial folder where your data files are located.
- 2. Enter **your name** and **today's date** in the comment section of the file, and then save the file as `orders.xml`. Figure 2-2 shows the contents of the `orders.xml` document for the first customer.

Figure 2-2

## First customer in the orders.xml document

```
<customer custID="cust201">
    <name title="Mr.">John Michael</name>
    <address>
        <![CDATA[
            41 West Plankton Avenue
            Orlando, FL 32820
        ]]>
    </address>
    <phone>(407) 555-3476</phone>
    <email>jk@example.net</email>
    <orders>
        <order orderID="or1089" orderBy="cust201">
            <orderDate>8/11/2017</orderDate>
            <items>
                <item itemNumber="WM100PL">
                    <itemPrice saleItem="N">39.95</itemPrice>
                    <itemQty>1</itemQty>
                </item>
                <item itemNumber="WM101P">
                    <itemPrice saleItem="Y">19.90</itemPrice>
                    <itemQty>2</itemQty>
                </item>
            </items>
        </order>
    </orders>
</customer>
```

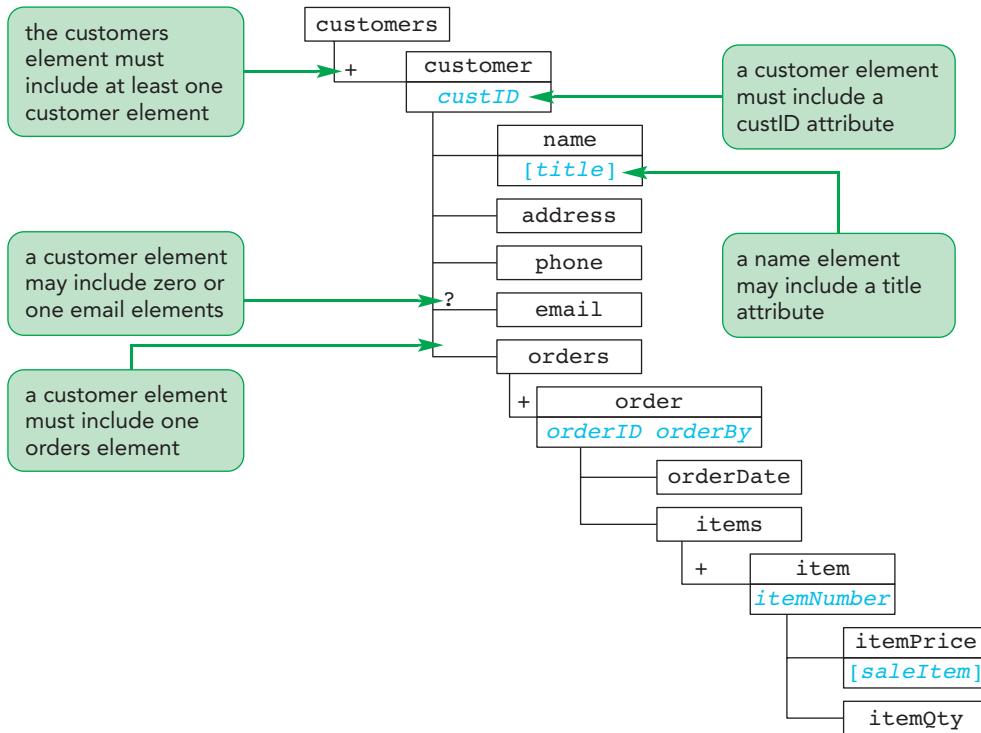
**Trouble?** You can complete the steps in this tutorial in any code editor or basic text editor. An XML editor such as Exchanger XML Editor can verify that your code is both well formed and valid. Many free general-purpose code editors, such as Notepad++ and Komodo Edit, can check that your code is well formed, and you can upload the code to an online service like <http://validator.w3.org> to confirm that the code is valid.

- 3. Compare the elements and attributes entered in the document with the table shown in Figure 2-1, noticing whether each piece of data is coded as an element or an attribute.
- 4. If you're using Exchanger XML Editor, click **XML** on the menu bar, and then click **Check Well-formedness**. This setting ensures that you don't see validation errors until your document is ready for validation.

Some elements in Benjamin's document, such as the `name` and `phone` elements, should appear only once for each customer, while other elements, such as the `order` and `item` elements, can appear multiple times. The `email` element is optional: Two customers have email addresses and one customer does not. The `itemPrice` and `itemQty` elements each appear once per `item` element.

Benjamin created the diagram shown in Figure 2-3 to better illustrate the structure of the elements and attributes in his document. Recall that the `+` symbol in front of an element indicates that at least one child element must be present in the document, and the `?` symbol indicates the presence of zero children or one child. Benjamin's diagram shows that the `customers`, `orders`, and `items` elements must have at least one `customer`, `order`, or `item` child, respectively, and that the `email` element is optional. Benjamin also indicated the presence of element attributes in blue below the relevant element names. An attribute name in square brackets ( [ ] ) is optional; all other attributes are required.

**Figure 2-3 Structure of the orders.xml document**



To keep accurate and manageable records, Benjamin must maintain this structure in his document. He wants to ensure that the customer information includes the address and phone number for each customer, the items each customer ordered, and the date each order was placed. In XML terms, this means that the document must be not only well formed, but also valid.

## Declaring a DTD

One way to create a valid document is to design a document type definition, or DTD, for the document. Recall that a DTD is a collection of rules that define the content and structure of an XML document. Used in conjunction with an XML parser that supports data validation, a DTD can:

- Ensure that all required elements are present in the document.
- Prevent undefined elements from being used in the document.
- Enforce a specific data structure on document content.
- Specify the use of element attributes and define their permissible values.
- Define default values for attributes.
- Describe how parsers should access non-XML or nontextual content.

A DTD is attached to an XML document by using a statement called a **document type declaration**, which is more simply referred to as a **DOCTYPE**. The DOCTYPE must be added to the document prolog after the XML declaration and before the document's root element. The purpose of the DOCTYPE is to either specify the rules of the DTD or provide information to the parser about where those rules are located. Each XML document can have only one DOCTYPE.

Because DTDs can be placed either within an XML document or in an external file, you can divide a DOCTYPE into two parts—an internal subset and an external subset. A DOCTYPE can include either or both of these parts. The **internal subset** contains the rules and declarations of the DTD placed directly into the document, using the form

```
<!DOCTYPE root
[  
    statements
]>
```

where *root* is the name of the document's root element and *statements* represents the declarations and rules of the DTD.

For example, the root element of the orders.xml document is customers, so a DOCTYPE in the orders.xml document has to specify *customers* as the value for the *root* parameter, as follows:

### TIP

The root value in a DOCTYPE must exactly match the name of the XML document's root element; otherwise, parsers will reject the document as invalid.

```
<!DOCTYPE customers
[  
    statements
]>
```

When the DTD is located in an external file, the DOCTYPE includes an **external subset** that indicates the location of the file. Locations can be defined using either a system identifier or a public identifier. With a **system identifier**, you specify the location of the DTD file. A DOCTYPE using a system identifier has the form

```
<!DOCTYPE root SYSTEM "uri">
```

where *root* is again the document's root element and *uri* is the URI of the external file. For example, if Benjamin placed the DTD for the orders.xml document in an external file named rules.dtd, he could access it using the following DOCTYPE:

```
<!DOCTYPE customers SYSTEM "rules.dtd">
```

**INSIGHT**

### Understanding URIs

The URI used in DOCTYPES looks like a web address used to create a link to a website; however, that is not its purpose. The purpose of a URI is simply to provide a unique string of characters that identifies a resource.

One version of a URI is the Uniform Resource Locator (URL), which is used to identify the location of a resource (such as a web page) on the web—for instance, <http://www.example.com>. One reason to use URLs as a basis for identifying DTDs is that URLs serve as a built-in mechanism on the web for generating unique addresses. If an XML vocabulary is made widely available, the DTD associated with that vocabulary needs to be unique. So, to ensure the uniqueness of any DTDs associated with the vocabularies developed for documents used by a specific company, it makes sense to use the company's web address as a foundation. Note that, although a URI doesn't actually need to point to a real site on the web, it is often helpful to place documentation at the site identified by the URI so users can go there to learn more about the XML vocabulary being referenced.

The use of URLs in declaring DTDs is widely accepted, but you can use almost any unique string identifier, such as MapFindsForYouModelNS or WM140PL. The main requirement is that a URI is unique so that it is not confused with the URIs of other DTDs.

When an XML vocabulary becomes widely used, developers seek to make the DTD easily accessible. This is done by creating a name for the DTD; this name is called a **public identifier** or a **formal public identifier**. The public identifier, which is optional, provides XML parsers with information about the DTD, including the owner or author of the DTD and the language in which the DTD is written. The syntax of a DOCTYPE that has only an external subset and involves a public identifier is

```
<!DOCTYPE root PUBLIC "id" "uri">
```

where *root* is the document's root element, *id* is the public identifier, and *uri* is the system location of the DTD. The system location is included in case the XML parser cannot process the document solely based on the information provided by the public identifier. In one sense, the public identifier acts like the namespace URI because it doesn't specify a physical location for the DTD; instead, it provides the DTD with a unique name that can be recognized by an XML parser. For example, XHTML documents that conform strictly to version 1.0 standards employ the following DOCTYPE:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

The public identifier is `-//W3C//DTD XHTML 1.0 Strict//EN`, a string of characters that XML parsers recognize as the identifier for the XHTML strict DTD. An XML parser that recognizes this public identifier can use it to try to retrieve the DTD associated with the XML vocabulary used in the document. If the parser cannot retrieve the DTD based on the public identifier, it can access it from the system location provided by the *uri* value, which is <http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>. As you can see, the system location acts as a backup to the public identifier. Most standard XML vocabularies such as XHTML and RSS have public identifiers. However, a customized XML vocabulary such as the one Benjamin created for Map Finds For You may not have a public identifier.



PROSKILLS

## Written Communication: Interpreting Public Identifiers

A public identifier is simply a public name given to a DTD that an XML parser can use to process and validate the document. The parser can use the public identifier to find the latest version of the DTD on the Internet. Each public identifier name has the structure

*standard//owner//description//language*

where *standard* indicates whether the DTD is a recognized standard, *owner* is the owner or developer of the DTD, *description* is a description of the XML vocabulary for which the DTD is developed, and *language* is a two-letter abbreviation of the language employed by the DTD. For example, the identifier for the XHTML 1.0 Strict vocabulary

*-//W3C//DTD XHTML 1.0 Strict//EN*

can be interpreted in the following manner:

- The initial – character tells the parser that the DTD is not a recognized standard. DTDs that are approved ISO (Internal Organization for Standardization) standards begin with the string ISO, whereas DTDs that are approved non-ISO standards begin with the + symbol.
- The next part of the id, w3c, indicates that this DTD is owned and developed by the W3C (World Wide web Consortium).
- The description content, DTD XHTML 1.0 Strict, specifies that this standard is used for the XHTML 1.0 Strict vocabulary.
- Finally, the closing EN characters indicate that the DTD is written in English.

If an XML parser has an internal catalog for working with public identifiers, it can use the information contained in the public identifier to work with the DTD. However, parsers are not required to work with public identifiers, so you should always include system locations for your public DTDs to support parsers that do not work with public identifiers.

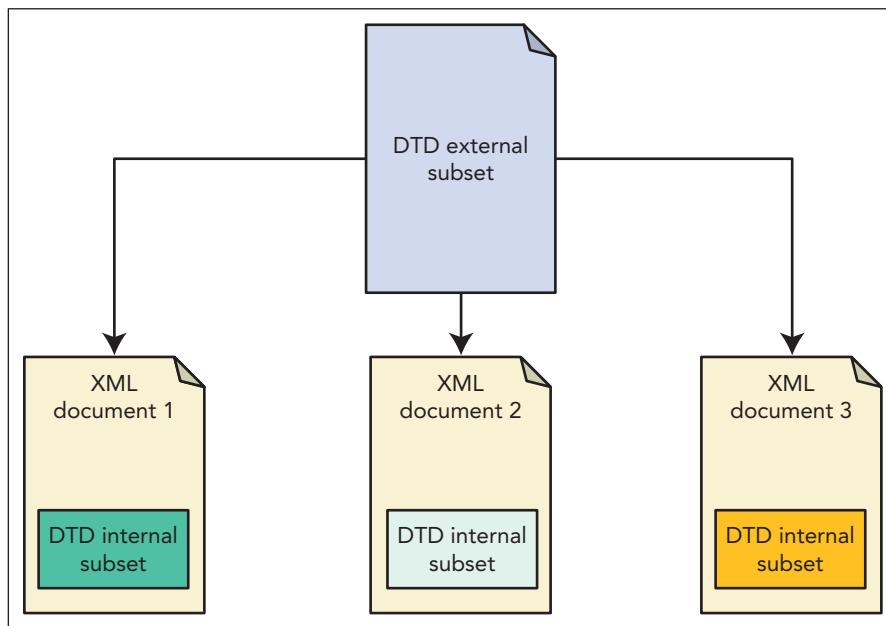
When writing code, you might create DOCTYPES with both internal and external subsets. A DTD that is shared among many different XML documents would be placed within an external file, whereas rules or declarations specific to an individual XML document would be placed within the internal subset. A DOCTYPE that combines both internal and external subsets and references a system identifier has the following form:

```
<!DOCTYPE root SYSTEM "uri"
[
  declarations
]>
```

If the DTD has a public identifier, then the DOCTYPE has the following form:

```
<!DOCTYPE root PUBLIC "id" "uri"
[
  declarations
]>
```

When a DOCTYPE contains both an internal and an external subset, the internal subset takes precedence over the external subset when conflict arises between the two. This is useful when an external subset is shared among several documents. The external subset would define some basic rules for all of the documents, and the internal subset would define rules that are specific to each document, as illustrated in Figure 2-4.

**Figure 2-4** Internal and external DTDs

In this way, internal and external DTDs work in the same manner as embedded and external style sheets. An XML environment composed of several documents and vocabularies might use both internal and external DTDs.

**REFERENCE*****Declaring a DTD***

- To declare an internal DTD subset, use the DOCTYPE

```
<!DOCTYPE root
[
  declarations
]>
```

where *root* is the name of the document's root element and *declarations* represents the statements that constitute the DTD.

- To declare an external DTD subset with a system location, use the DOCTYPE

```
<!DOCTYPE root SYSTEM "uri">
```

where *uri* is the URI of the external DTD file.

- To declare an external DTD subset with a public location, use the DOCTYPE

```
<!DOCTYPE root PUBLIC "id" "uri">
```

where *id* is the public identifier of the DTD.

## Writing the Document Type Declaration

Benjamin wants you to place the DTD directly in his XML document so he can easily compare the DTD to the document content. You'll start by inserting a blank DOCTYPE into the orders.xml file.

The DOCTYPE must be located within the prolog, which is before the document body.

### To insert a DOCTYPE into the orders.xml document:

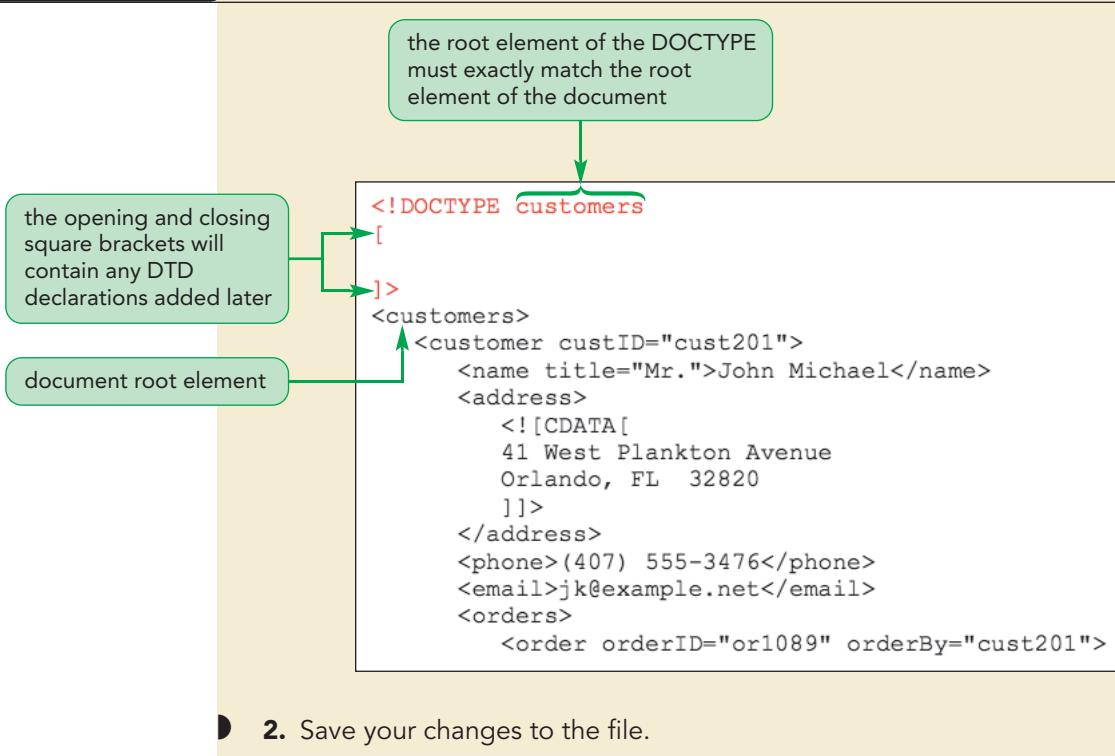
- 1. Directly above the opening <customers> tag, insert the following DOCTYPE:

```
<!DOCTYPE customers
[
]>
```

Figure 2-5 shows the updated document.

Figure 2-5

Blank DOCTYPE inserted



- 2. Save your changes to the file.

Now that you've added a DOCTYPE to Benjamin's document, you can begin adding statements to the DTD to define the structure and content of his XML vocabulary.

## Declaring Document Elements

In a valid document, every element must be declared in the DTD. An element type declaration, or element declaration, specifies an element's name and indicates what content the element can contain. It can even specify the order in which elements appear in the document. The syntax of an element declaration is

### TIP

Element declarations must begin with <!ELEMENT in all uppercase letters, and not <!Element or <element.

```
<!ELEMENT element content-model>
```

where *element* is the name of the element and *content-model* specifies what type of content the element contains. The element name is case sensitive, so if the element name is *VENDORS*, it must be entered as *VENDORS* (not *Vendors* or *vendors*) in the element declaration. Remember that element names cannot contain any spaces or

reserved symbols such as < or >. The *content-model* value can be one of three specific keywords (ANY, EMPTY, #PCDATA), or one of two content descriptions (sequence, #PCDATA with sequence), as follows:

- **ANY**: The element can store any type of content or no content at all.
- **EMPTY**: The element cannot store any content.
- **#PCDATA**: The element can contain only parsed character data.
- **Sequence**: The element can contain only child elements.
- **#PCDATA with sequence**: The element can store both parsed character data and child elements.

Generally, elements contain parsed character data or child elements. For example, in Benjamin's document, the `phone` element contains a text string that stores a customer's phone number, which is parsed character data. On the other hand, the `customer` element contains five child elements (`name`, `address`, `phone`, `email`, and `orders`). The following sections explore the five content types in more detail.

## Elements Containing Any Type of Content

The most general type of content model is ANY, which allows an element to store any type of content. The syntax to allow any element content is

```
<!ELEMENT element ANY>
```

For example, the declaration

```
<!ELEMENT vendor ANY>
```

in the DTD would allow the `vendor` element to contain any type of content, or none at all. Any of the following content in the XML document would satisfy this element declaration:

```
<vendor>V12300 Mapping Down the Road</vendor>
```

or

```
<vendor />
```

or

```
<vendor>
  <number>PLBK70</number>
  <name>Mapping Down the Road</name>
</vendor>
```

Allowing an element to contain any type of content has little value in document validation. After all, the idea behind validation is to enforce a particular set of rules on elements and their content; allowing any content defeats the purpose of these rules.

## Empty Elements

The EMPTY content model is reserved for elements that store no content. The syntax for an empty element declaration is

```
<!ELEMENT element EMPTY>
```

The element declaration

```
<!ELEMENT shelf EMPTY>
```

would require the `shelf` element to be entered as an empty element, as follows:

```
<shelf />
```

Including content in an element that uses the EMPTY content model would cause XML parsers to reject the document as invalid.

## Elements Containing Parsed Character Data

Recall that parsed character data, or PCDATA, is text that is parsed by a parser. The #PCDATA content model value is reserved for elements that can store parsed character data, which are declared as follows:

```
<!ELEMENT element (#PCDATA)>
```

For example, the declaration

```
<!ELEMENT name (#PCDATA)>
```

permits the following element in an XML document:

```
<name>John Michael</name>
```

An element declaration employing the #PCDATA content model value does not allow for child elements. As a result, the `name` element

### Not valid code:

```
<name>
  <first>John</first>
  <last>Michael</last>
</name>
```

is not considered valid because child elements are not considered parsed character data.

### REFERENCE

#### Specifying Types of Element Content

- To declare an element that may contain any type of content, insert the declaration

```
<!ELEMENT element ANY>
```

where `element` is the element name.

- To declare an empty element containing no content whatsoever, use the following declaration:

```
<!ELEMENT element EMPTY>
```

- To declare an element that may contain only parsed character data, use the following declaration:

```
<!ELEMENT element (#PCDATA)>
```

The `name`, `address`, `phone`, `email`, `orderDate`, `itemPrice`, and `itemQty` elements in the `orders.xml` document contain only parsed character data. You'll add declarations for these elements to the DTD.

#### To declare elements containing parsed character data in the `orders.xml` document:

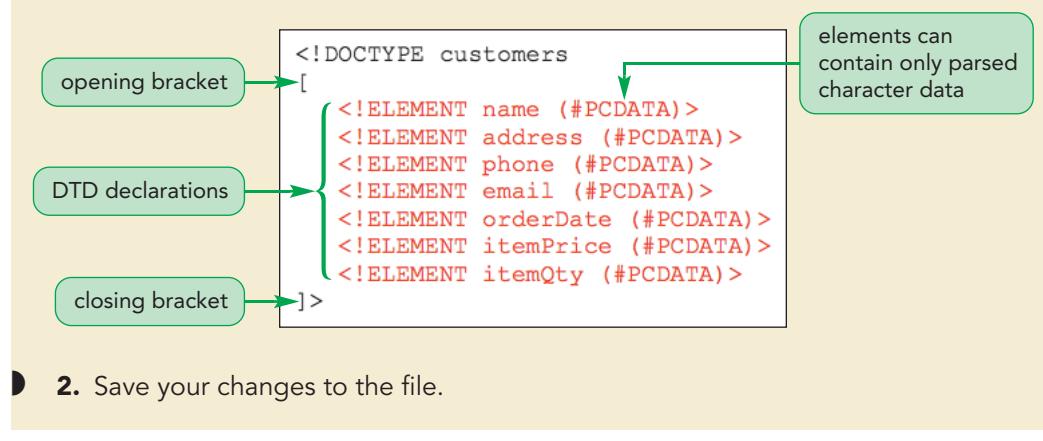
- Within the DOCTYPE, insert the following element declarations:

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT orderDate (#PCDATA)>
<!ELEMENT itemPrice (#PCDATA)>
<!ELEMENT itemQty (#PCDATA)>
```

Figure 2-6 shows the updated DOCTYPE.

**Figure 2-6**

### Element declarations



## Working with Child Elements

Next, you'll consider how to declare an element that contains only child elements. The syntax for such a declaration is

```
<!ELEMENT element (children)>
```

where *element* is the parent element and *children* is a listing of its child elements. The simplest form for the listing consists of a single child element associated with a parent. For example, the declaration

```
<!ELEMENT customer (phone)>
```

indicates that the *customer* element can contain only a single child element named *phone*. The following code would be invalid under this element declaration because the *customer* element contains two child elements—*name* and *phone*:

```
<customer>
    <name>John Michael</name>
    <phone>(407) 555-3476</phone>
</customer>
```

For content that involves multiple child elements, you can specify the elements in a sequence or you can specify a choice of elements.

### Specifying an Element Sequence

A sequence is a list of elements that follow a defined order. The syntax to specify child elements in a sequence is

```
<!ELEMENT element (child1, child2, ...)>
```

where *child1*, *child2*, etc., represents the sequence of child elements within the parent. The order of the child elements in an XML document must match the order defined in the element declaration. For example, the following element declaration defines a sequence of three child elements for each *customer* element:

```
<!ELEMENT customer (name, phone, email)>
```

Under this declaration, the following code is valid:

```
<customer>
  <name>John Michael</name>
  <phone>(407) 555-3476</phone>
  <email>jk@example.net</email>
</customer>
```

However, even though the elements and their contents are identical in the following code, the code is not valid because the sequence doesn't match the defined order:

```
<customer>
  <name>John Michael</name>
  <email>jk@example.net</email>
  <phone>(407) 555-3476</phone>
</customer>
```

## Specifying an Element Choice

Rather than defining a sequence of child elements, the element declaration can define a choice of possible elements. The syntax used to specify an element choice is

```
<!ELEMENT element (child1 | child2 | ...)>
```

where *child1*, *child2*, etc., are the possible child elements of the parent. For example, the following declaration allows the `customer` element to contain either the `name` or the `company` element:

```
<!ELEMENT customer (name | company)>
```

Based on this declaration, either of the following code samples is valid:

```
<customer>
  <name>John Michael</name>
</customer>
or
<customer>
  <company>Mapping Down the Road</company>
</customer>
```

However, under this declaration, a `customer` element cannot include both the `name` and `company` elements because the choice model allows only one of the child elements listed.

An element declaration can combine both a sequence and a choice of child elements. For example, the following declaration limits the `customer` element to three child elements, the first of which is either the `name` or the `company` element, followed by the `phone` and then the `email` element:

```
<!ELEMENT customer ((name | company), phone, email)>
```

Under this declaration, both of the following code samples are valid:

```
<customer>
  <name>Lea Ziegler</name>
  <phone>(813)555-8931</phone>
  <email>LZiegler@example.net</email>
</customer>
or
<customer>
  <company>VTech Productions</company>
  <phone>(813)555-8931</phone>
  <email>LZiegler@example.net</email>
</customer>
```

However, a `customer` element that does not start with either a `name` element or a `company` element followed by the `phone` and `email` elements would be invalid.

## REFERENCE

**Specifying Child Elements**

- To specify the sequence of child elements, use the declaration  
`<!ELEMENT element (child1, child2, ...)>`  
 where `child1, child2, ...` is the order in which the child elements must appear within the parent element.
- To allow for a choice of child elements, use the declaration  
`<!ELEMENT element (child1 | child2 | ...)>`  
 where `child1, child2, ...` are the possible children of the parent element.
- To combine a choice and a sequence of child elements in the same declaration, specify the choice elements within an additional set of parentheses, in the appropriate place within the sequence, as in the following code:  
`<!ELEMENT element ((child1 | child2 | ...), child3, child4, ...)>`

**Modifying Symbols**

So far, all the content models you have seen assume that each child element occurs once within its parent. If you need to specify duplicates of the same element, you could repeat the element name in the list. For example, the following element declaration indicates that the `customer` element must contain two `phone` elements:

```
<!ELEMENT customer (phone, phone)>
```

## TIP

You can specify that an element contains a minimum number of a child element by entering duplicate elements equal to the minimum number and adding a `+` to the last one.

However, it's rare that you specify the exact number of duplicate elements. Instead, DTDs use more general numbering with a modifying symbol that specifies the number of occurrences of each element. The three modifying symbols are the question mark (`?`), the plus sign (`+`), and the asterisk (`*`). These are the same symbols you saw when creating a tree diagram for an XML document. As before, the `?` symbol indicates that an element occurs zero times or one time, the `+` symbol indicates that an element occurs at least once, and the `*` symbol indicates that an element occurs zero times or more. There are no other modifying symbols. If you want to specify an exact number of child elements, such as the two `phone` elements discussed above, you must repeat the element name the appropriate number of times.

In the `orders.xml` document, the `customers` element must contain at least one element named `customer`. The element declaration for this is

```
<!ELEMENT customers (customer+)>
```

As this code demonstrates, a modifying symbol is placed directly after the element it modifies. You can also include modifying symbols in element sequences. For example, in Benjamin's document, each `customer` element contains the `name`, `address`, `phone`, and `email` elements, but the `email` element is optional, occurring either zero times or one time. The element declaration for this is

```
<!ELEMENT customer (name, address, phone, email?)>
```

The three modifying symbols can also modify entire element sequences or choices. You do this by placing the character immediately following the closing parenthesis of the sequence or choice. For example, the declaration

```
<!ELEMENT order (orderDate, items)+>
```

indicates that the child element sequence (`orderDate`, `items`) can be repeated one or more times within each `order` element. Of course, each time the sequence is repeated, the `orderDate` element must appear first, followed by the `items` element.

When applied to a choice model, the modifying symbols allow for multiple combinations of each child element. The declaration

```
<!ELEMENT customer (name | company)+>
```

allows any of the following lists of child elements:

- `name`
- `company`
- `name, company`
- `name, name, company`
- `name, company, company, name`

The only requirement is that the combined total of `name` and `company` elements be greater than zero.

## REFERENCE

### Applying Modifying Symbols

- To specify that an element can appear zero times or one time, use `item?` in the element declaration in the DTD, where `item` is an element name or a sequence or choice of elements.
- To specify one or more occurrences of an item, use `item+`
- To specify zero or more occurrences of an item, use `item*`

Now that you've seen how to specify the occurrences of child elements, you'll add these declarations to the DTD for Benjamin's document. The declarations can be entered in any order, but you'll insert the declarations in the order in which the elements appear in the document. You'll also insert blank lines between groups of declarations to make the code easier to read.

### To declare the child elements:

1. In the DTD of the `orders.xml` file, on the line below the opening bracket, insert the following element declaration, followed by a blank line:
 

```
<!ELEMENT customers (customer+)>
```

 This declaration specifies that the root `customers` element must contain at least one `customer` element.
2. Above the `name` declaration, insert the following declaration followed by a blank line:
 

```
<!ELEMENT customer (name, address, phone, email?, orders)>
```

 This declaration specifies that a `customer` element must contain, in order, a `name` element, an `address` element, and a `phone` element; it may then contain an `email` element; and then it must contain an `orders` element.
3. Below the declaration for the `email` element, insert the following declaration:
 

```
<!ELEMENT orders (order+)>
```

This declaration specifies that an `orders` element must contain at least one `order` element.

- 4. Below the `orders` declaration, insert a blank line followed by the following declaration:

```
<!ELEMENT order (orderDate, items)>
```

This declaration specifies that each `order` element must contain an `orderDate` element and an `items` element.

- 5. Below the `orderDate` declaration, insert the following declaration:

```
<!ELEMENT items (item+)>
```

This declaration specifies that each `items` element must contain at least one `item` element.

- 6. Below the `items` declaration, insert a blank line followed by the following declaration:

```
<!ELEMENT item (itemPrice, itemQty)>
```

This declaration specifies that the `item` element must contain the `itemPrice` and `itemQty` elements.

- 7. Add a blank line after the `name` declaration, the `order` declaration, and the `item` declaration. Figure 2-7 shows the completed element declarations in the DTD.

Figure 2-7

### Declarations for the child elements

customers element  
must contain one  
or more customer  
elements

orders element  
must contain one or more  
order elements

items element  
must contain one or more  
item elements

```
<!DOCTYPE customers
[
    <!ELEMENT customers (customer+)>

    <!ELEMENT customer (name, address, phone, email?, orders)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT address (#PCDATA)>
        <!ELEMENT phone (#PCDATA)>
        <!ELEMENT email (#PCDATA)>
        <!ELEMENT orders (order+)>
            <!ELEMENT order (orderDate, items)>
                <!ELEMENT orderDate (#PCDATA)>
                <!ELEMENT items (item+)>
                    <!ELEMENT item (itemPrice, itemQty)>
                        <!ELEMENT itemPrice (#PCDATA)>
                        <!ELEMENT itemQty (#PCDATA)>
]
```

customer element must contain the  
name, address, and phone elements  
in that order; may then contain the  
email element; and then must  
contain the orders element

order element must contain  
the orderDate element and  
then the items element

item element must contain  
the itemPrice element and  
then the itemQty element

- 8. Save your changes to the file.

To see how these element declarations represent the structure of the document, compare Figure 2-7 with the tree diagram shown earlier in Figure 2-3.

**INSIGHT*****DTDs and Mixed Content***

An XML element is not limited to either parsed character data or child elements. If an element contains both, its content is known as **mixed content**. For example, the `title` element in the following code contains both the text of the title and a collection of `subtitle` elements:

```
<title>The Adventures of Sherlock Holmes
  <subtitle>The Sign of Four</subtitle>
  <subtitle>by Sir Arthur Conan Doyle</subtitle>
</title>
```

To declare mixed content in a DTD, you use the following declaration:

```
<!ELEMENT element (#PCDATA | child1 | child2 | ...)*>
```

This declaration applies the `*` modifying symbol to a choice of parsed character data or child elements. Because the `*` symbol is used with a choice list, the element can contain any number of occurrences of child elements or text strings of parsed character data, or it can contain no content at all. For example, the declaration

```
<!ELEMENT title (#PCDATA | subtitle)*>
<!ELEMENT subtitle (#PCDATA)>
```

allows the `title` element to contain any number of text strings of parsed character data interspersed by `subtitle` elements. The `subtitle` elements themselves can contain only parsed character data.

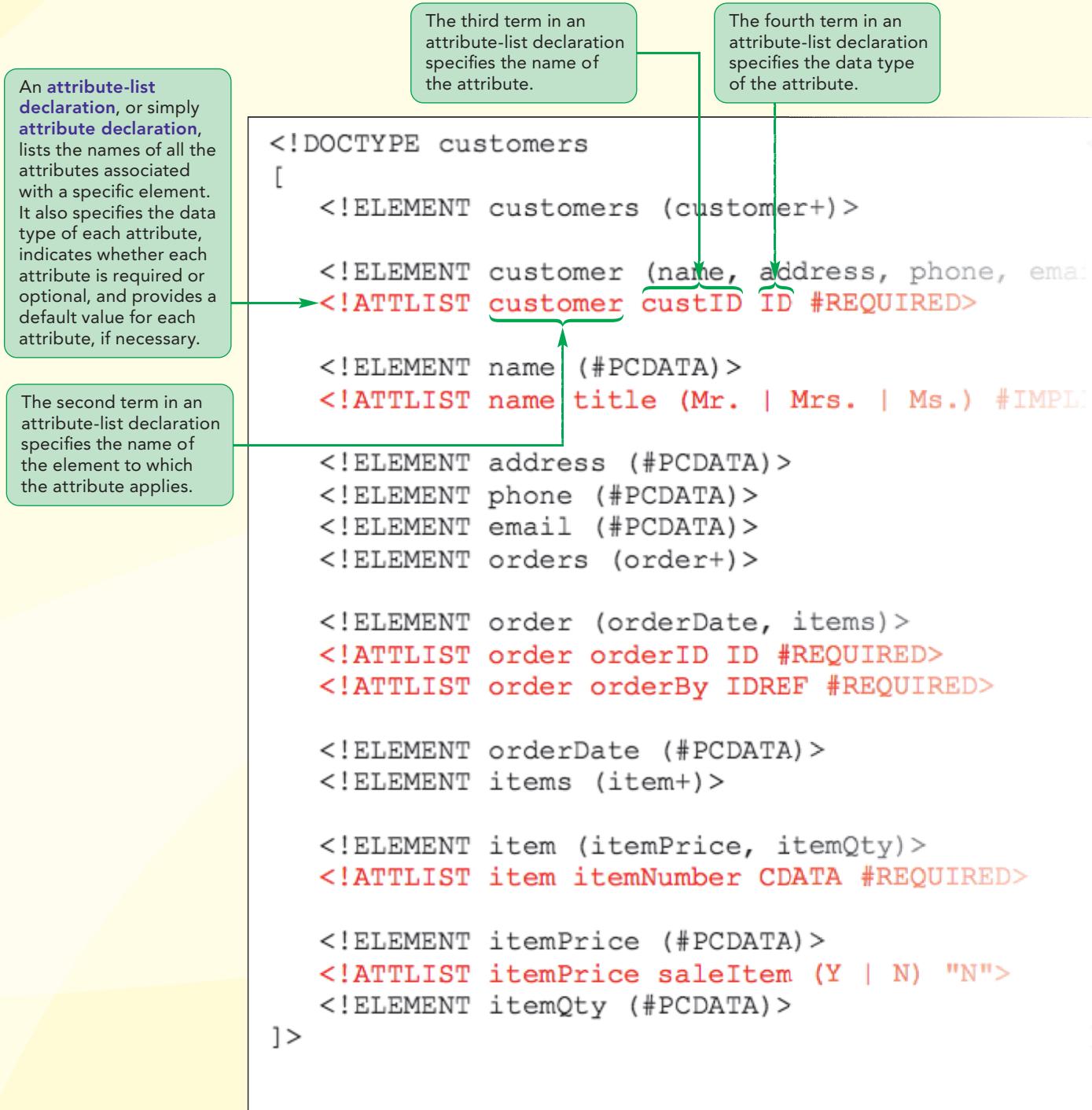
Because they are very flexible, elements with mixed content do not add much defined structure to a document. You can specify only the names of the child elements, and you cannot constrain the order in which those child elements appear or control the number of occurrences for each element. An element might contain only text or it might contain any number of child elements in any order. For this reason, it is best to avoid working with mixed content if you want a tightly structured document.

At this point you've defined a structure for the elements in the `orders.xml` file. You will declare the attributes associated with those elements in the next session.

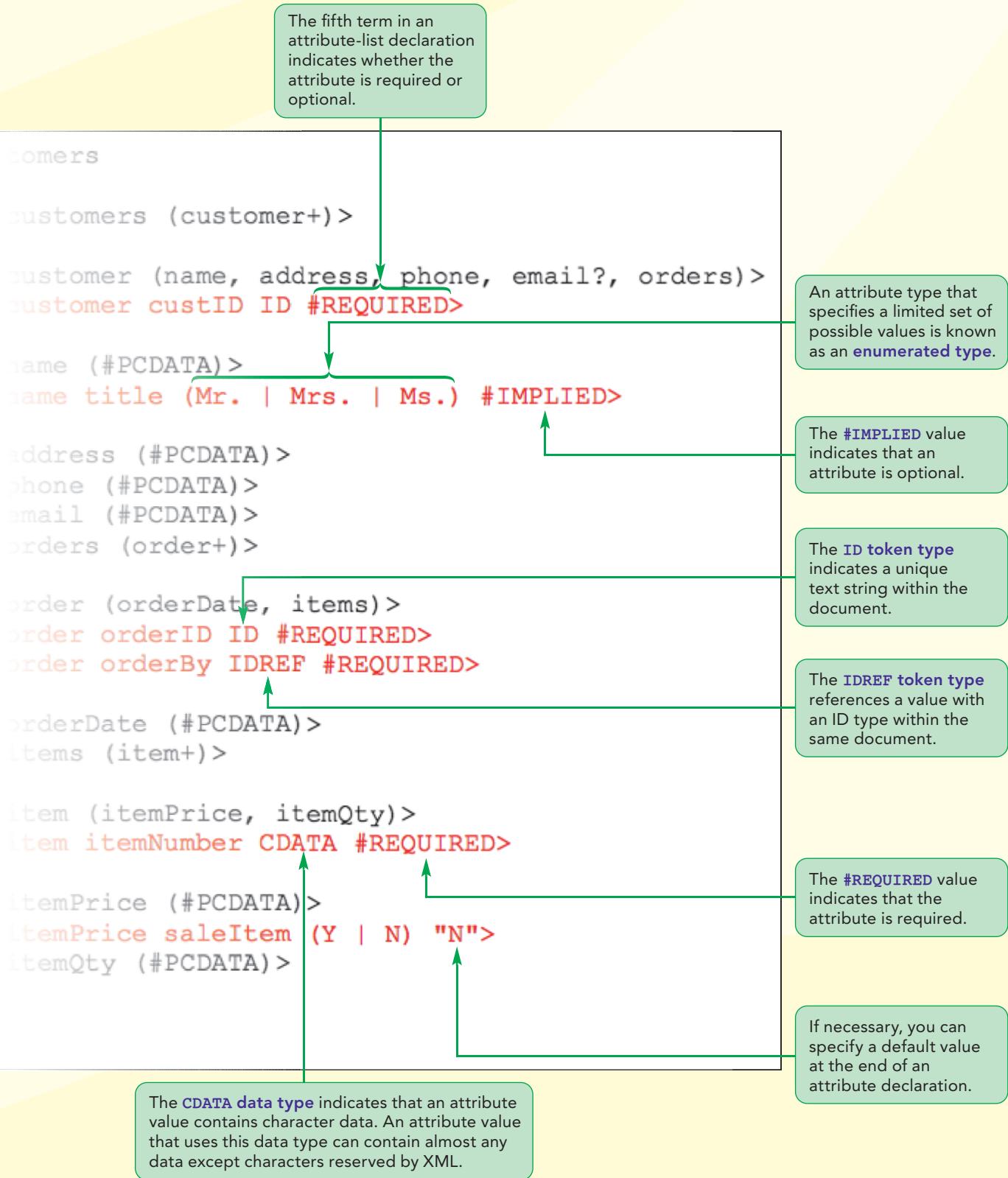
**REVIEW*****Session 2.1 Quick Check***

1. What code would you enter to connect an XML document with the root element `Inventory` to a DTD stored in the file `books.dtd`?
2. What declaration would you enter to allow the `book` element to contain any content?
3. What declaration would you enter to specify that the `video` element is empty?
4. What declaration would you enter to indicate that the `book` element can contain only parsed character data?
5. What declaration would you enter to indicate that the `book` element can contain only a single child element named `author`?
6. What declaration would you enter to indicate that the `book` element can contain one or more child elements named `author`?
7. What declaration would you enter to allow the `part` element to contain a sequence that begins with a choice of the `partNum` or `partName` child elements, followed by child elements named `description` and then `price`?

## Session 2.2 Visual Overview:



# Defining Attributes within a DTD



## Declaring Attributes

In the previous session, you defined the structure of Benjamin's document by declaring all the elements in the orders.xml document and indicating what type of content each element could contain. However, the tree structure shown in Figure 2-3 also includes attributes for some elements. For the document to be valid, you must also declare all the attributes associated with the elements. You must indicate whether each attribute is required or optional, and you must indicate what kinds of values are allowed for each attribute. Finally, you can indicate whether each attribute has a default value associated with it. Figure 2-8 describes all the attributes that Benjamin intends to use in the orders.xml document along with the properties of each attribute.

**Figure 2-8** Properties of attributes used in orders.xml

Element	Attribute	Description	Required?	Allowable Values
<b>customer</b>	custID	customer ID number	Yes	character data
<b>name</b>	title	title associated with the customer's name	No	"Mr.", "Mrs.", or "Ms."
<b>order</b>	orderID	order ID number	Yes	character data
	orderBy	ID of the customer placing the order	Yes	character data
<b>item</b>	itemNumber	item number	Yes	character data
<b>itemPrice</b>	saleItem	whether item price is a sale price	No	"Y" or "N" (default)

For example, every **customer** element must have a **custID** attribute to record the customer ID value. If a **custID** attribute is omitted from a **customer** element, the document is invalid.

To enforce attribute properties, you must add an attribute-list declaration to the document's DTD for each element that includes attributes. An attribute-list declaration:

- lists the names of all the attributes associated with a specific element
- specifies the data type of each attribute
- indicates whether each attribute is required or optional
- provides a default value for each attribute, if necessary

The syntax for declaring a list of attributes is

```
<!ATTLIST element attribute1 type1 default1
            attribute2 type2 default2
            ... >
```

where **element** is the name of the element associated with the attributes; **attribute1**, **attribute2**, etc., are the names of attributes; **type1**, **type2**, etc., are the attributes' data types; and **default1**, **default2**, etc., indicate whether each attribute is required and whether it has a default value. In practice, declarations for elements with multiple attributes are often easier to interpret if the attributes are declared separately rather than in one long declaration. The following is an equivalent form in the DTD:

```
<!ATTLIST element attribute1 type1 default1>
<!ATTLIST element attribute2 type2 default2>
...

```

XML parsers combine the different statements into a single attribute-list declaration. If a processor encounters more than one declaration for the same attribute, it ignores the second statement. Attribute-list declarations can be located anywhere within the document type declaration; however, it is often easiest to work with attribute declarations that are located adjacent to the declaration for the element with which they are associated.

### Declaring Attributes in a DTD

- To declare a list of attributes associated with an element, enter the declaration

```
<!ATTLIST element attribute1 type1 default1
            attribute2 type2 default2
            ...>
```

or

```
<!ATTLIST element attribute1 type1 default1>
<!ATTLIST element attribute2 type2 default2>
...
```

where *element* is the element associated with the attributes; *attribute1*, *attribute2*, etc., are the names of attributes; *type1*, *type2*, etc., are the attributes' data types; and *default1*, *default2*, etc., indicate whether each attribute is required and whether it has a default value.

- To indicate that an attribute contains character data, use

*attribute* CDATA

where *attribute* is the name of the attribute.

- To constrain an attribute value to a list of possible values, use

*attribute* (*value1* | *value2* | *value3* | ...)

where *value1*, *value2*, etc., are allowed values for the attribute.

- To indicate that an attribute contains ID values, use

*attribute* ID

- To indicate that an attribute contains a white space-separated list of ID values, use

*attribute* IDS

- To indicate that an attribute contains a reference to an ID value, use

*attribute* IDREF

- To indicate that an attribute contains a white space-separated list of references to ID values, use

*attribute* IDREFS

- To constrain an attribute to an XML name containing only letters, numbers, and the punctuation symbols underscore (\_), hyphen (-), period (.), and colon (:), but no white space, use

*attribute* NMOKEN

- To constrain an attribute to a white space-separated list of XML names, use

*attribute* NMOKENS

As a first step in adding attributes to the DTD for the orders.xml document, you will declare the names of the attributes and the elements with which they are associated in the document.

*Note: Because you are not yet going to include the data type and default values for the attributes, these attribute declarations are incomplete and would be rejected by any XML parser. You'll fix this later in this session.*

#### To declare the attributes in the orders.xml document:

- If you took a break after the previous session, make sure the orders.xml file is open in your editor.

- 2. Insert a new line below the `customer` element declaration, enter the following declaration, and then insert a new line:

```
<!ATTLIST customer custID>
```

This declaration indicates that the `customer` element can contain an attribute named `custID`.

- 3. Insert a new line below the declaration for the `name` element, enter the following declaration, and then insert a new line:

```
<!ATTLIST name title>
```

This declaration indicates that the `name` element can contain a `title` attribute.

- 4. On the blank line below the declaration for the `order` element, insert the following two declarations on new lines, and then insert a new line:

```
<!ATTLIST order orderID>
```

```
<!ATTLIST order orderBy>
```

These declarations indicate that the `order` element can contain attributes named `orderID` and `orderBy`.

- 5. On the blank line below the declaration for the `item` element, enter the following attribute declaration, and then insert a blank line:

```
<!ATTLIST item itemNumber>
```

This declaration indicates that the `item` element can contain an attribute named `itemNumber`.

- 6. Insert a blank line below the declaration for the `itemPrice` element, and then enter the following declaration:

```
<!ATTLIST itemPrice saleItem>
```

This declaration indicates that the `itemPrice` element can contain an attribute named `saleItem`. Figure 2-9 shows the revised code including the attribute-list declarations.

Figure 2-9

## Declarations for the attribute names

you can place an attribute-list declaration anywhere within the DOCTYPE, but placing it after the relevant element declaration helps organize your code

```
<!DOCTYPE customers
[
    <!ELEMENT customers (customer+)

    <!ELEMENT customer (name, address, phone, email?, orders)
    <!ATTLIST customer custID>

    <!ELEMENT name (#PCDATA)
    <!ATTLIST name title>

    <!ELEMENT address (#PCDATA)>
    <!ELEMENT phone (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT orders (order+)

    <!ELEMENT order (orderDate, items)
    <!ATTLIST order orderID>
    <!ATTLIST order orderBy>

    <!ELEMENT orderDate (#PCDATA)>
    <!ELEMENT items (item+)

    <!ELEMENT item (itemPrice, itemQty)
    <!ATTLIST item itemNumber>

    <!ELEMENT itemPrice (#PCDATA)>
    <!ATTLIST itemPrice saleItem>
    <!ELEMENT itemQty (#PCDATA)>
]>
```

## 7. Save your changes to the file.

**Trouble?** When you save changes, your XML editor may indicate that the document contains an error and is no longer well formed. This is because the attribute-list declarations you entered are not yet complete. You'll fix this error by completing the declarations in the steps that follow.

## Working with Attribute Types

The next step in defining these attributes is to specify the type of data each attribute can contain. Attribute values can consist only of character data, but you can control the format of those characters. Figure 2-10 lists the different data types that DTDs support for attribute values. Each data type gives you a varying degree of control over an attribute's content. You will investigate each of these types in greater detail, starting with character data.

**Figure 2-10** Attribute types

Attribute Value	Description
CDATA	Any character data except characters reserved by XML
enumerated list	A list of possible attribute values
ID	A unique text string
IDREF	A reference to an ID value
IDREFS	A list of ID values separated by white space
ENTITY	A reference to an external unparsed entity
ENTITIES	A list of entities separated by white space
NMOKEN	An accepted XML name
NMOKENS	A list of XML names separated by white space
NOTATION	The name of a notation defined in the DTD

## Character Data

Attribute values specified as character data (CDATA) can contain almost any data except characters reserved by XML for other purposes, such as <, >, and &. To declare an attribute value as character data, you add the **CDATA** data type to the attribute declaration with the syntax

### TIP

You'll learn more about default attribute values later in this tutorial.

```
<!ATTLIST element attribute CDATA default>
```

where the optional term **default** specifies a default value. For example, the item number of each item in Benjamin's document is expressed in character data. To indicate this in the DTD, you would add the **CDATA** attribute type to the declaration for the **itemNumber** attribute as follows:

```
<!ATTLIST item itemNumber CDATA>
```

Any of the following attribute values are allowed under this declaration because they all contain character data:

```
<item itemNumber="340-978"> ... </item>
<item itemNumber="WMPro"> ... </item>
<item itemNumber="WM101PL"> ... </item>
```

In Benjamin's document, the attribute **itemNumber** contains character data. You'll add this information to its attribute declaration.

### To specify that the **itemNumber** attribute contains character data:

- 1. In the **orders.xml** file, within the attribute-list declaration for **itemNumber**, just before the closing **>**, type a **space**, and then type **CDATA** as shown in Figure 2-11.

Figure 2-11

**Character data type added to the attribute-list declaration**

```

<!ELEMENT orderDate (#PCDATA)>
<!ELEMENT items (item+)>

<!ELEMENT item (itemPrice, itemQty)>
<!ATTLIST item itemNumber CDATA>

<!ELEMENT itemPrice (#PCDATA)>
<!ATTLIST itemPrice saleItem>
<!ELEMENT itemQty (#PCDATA)>
]>

```

specifies that attribute values are composed of character data

- 2. Save your changes to the file.

Even though you may work with attribute values that will use only numbers as data, it's not possible to declare that an attribute's values must contain only a certain type of characters, such as integers or numbers. To indicate that an attribute value must be an integer or a number, you use schemas—a topic you will study in a later tutorial.

## Enumerated Types

The CDATA data type allows for almost any string of characters, but in some cases you want to restrict an attribute to a set of possible values. For example, Benjamin uses the `title` attribute to indicate the title by which a customer chooses to be addressed. He needs to restrict the `title` attribute to values of "Mr.", "Mrs.", or "Ms." An attribute type that specifies a limited set of possible values is known as an enumerated type. The general form of an attribute declaration that uses an enumerated type is

```
<!ATTLIST element attribute (value1 | value2 | value3 | ...)  
default >
```

where `value1`, `value2`, etc., are allowed values for the specified attribute. To limit the value of the `title` attribute to "Mr." or "Mrs." or "Ms.", Benjamin can include an enumerated type, as in the following declaration:

```
<!ATTLIST name title (Mr. | Mrs. | Ms.)>
```

Under this declaration, any `title` attribute whose value is not "Mr.", "Mrs.", or "Ms." causes parsers to reject the document as invalid. You'll add this enumerated type to the DTD along with an enumerated type to specify the enumerated values for the `title` attribute of the `name` element.

### To declare enumerated data types for the two `title` attributes:

- 1. Within the `title` attribute declaration for the `name` element, just before the closing `>`, type a **space** and then type **(Mr. | Mrs. | Ms.)**.
- 2. Within the `saleItem` attribute declaration for the `itemPrice` element, just before the closing `>`, type a **space** and then type **(Y | N)**. Figure 2-12 shows the revised DTD.

Figure 2-12

## Attributes with enumerated data types

```

<!DOCTYPE customers
[ 
  <!ELEMENT customers (customer+)>

  <!ELEMENT customer (name, address, phone, email?, orders)>
  <!ATTLIST customer custID>

  <!ELEMENT name (#PCDATA)>
  <!ATTLIST name title (Mr. | Mrs. | Ms.)>
  values of the title attribute are limited to Mr., Mrs., or Ms.

  <!ELEMENT address (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT orders (order+)>

  <!ELEMENT order (orderDate, items)>
  <!ATTLIST order orderID>
  <!ATTLIST order orderBy>

  <!ELEMENT orderDate (#PCDATA)>
  <!ELEMENT items (item+)>

  <!ELEMENT item (itemPrice, itemQty)>
  <!ATTLIST item itemNumber CDATA>
  values of the saleItem attribute are limited to Y or N

  <!ELEMENT itemPrice (#PCDATA)>
  <!ATTLIST itemPrice saleItem (Y | N)>
  <!ELEMENT itemQty (#PCDATA)>
]>

```

- 3. Save your changes to the file.

Another type of enumerated type is a notation. A **notation** associates the value of an attribute with a `<!NOTATION>` declaration that is inserted elsewhere in the DTD. Notations are used when an attribute value refers to a file containing nontextual data, such as a graphic image or a video clip. You will learn more about notations and how to work with nontextual data in the next session.

## Tokenized Types

**Tokenized types** are character strings that follow certain specified rules for format and content; these rules are known as **tokens**. DTDs support four kinds of tokens—ID, ID reference, name token, and entity.

An **ID token** is used when an attribute value must be unique within a document. In Benjamin's document, the `customer` element contains the `custID` attribute, which stores a unique ID for each customer. To prevent users from entering the same `custID` value for different customers, Benjamin can define the attribute type for the `custID` attribute as follows:

```
<!ATTLIST customer custID ID>
```

Under this declaration, the following elements are valid:

```
<customer custID="cust201"> ... </customer>
<customer custID="cust202"> ... </customer>
```

However, the following elements occurring in the same document would not be valid because the same `custID` value is used more than once:

```
<customer custID="cust201"> ... </customer>
<customer custID="cust201"> ... </customer>
```

When an ID value is declared in a document, other attribute values can reference it using the `IDREF` token. An attribute declared using the **IDREF token** must have a value equal to the value of an ID attribute located somewhere in the same document. This enables an XML document to contain cross-references between one element and another.

For example, the `order` element in Benjamin's document has an attribute named `orderBy`, which contains the ID of the customer who placed the order. Assuming the `custID` value of the `customer` element is defined using the ID token, Benjamin can ensure that the `orderBy` value refers to an actual customer by using the following declaration:

```
<!ATTLIST order orderBy IDREF>
```

When an XML parser encounters this attribute, it searches the XML document for an ID value that matches the value of the `orderBy` attribute. If one of the attribute values doesn't have a matching ID value in the document, the parser rejects the document as invalid. However, you cannot specify that an XML parser limit its search to only particular elements or attributes. Any attribute that has been declared by the data type ID is a candidate for an ID reference.

An attribute can contain multiple IDs and ID references in a list, with entries separated by white space. For example, Benjamin might list all the orders made by a certain customer as an attribute of the `customer` element as in the following sample code:

```
<customer orders="or1089 or1021 or1122">
  ...
  <order orderID="or1089"> ... </order>
  <order orderID="or1021"> ... </order>
  <order orderID="or1122"> ... </order>
  ...
</customer>
```

Each ID listed in the `orders` attribute must match an ID value located elsewhere in the document. If one does not, Benjamin would want the document to be declared invalid.

To declare that an attribute contains a list of IDs, you apply the `IDS` attribute type to the attribute declaration as follows:

```
<!ATTLIST element attribute IDS default>
```

To declare that an attribute contains a list of ID references, you use the `IDREFS` attribute type as follows:

```
<!ATTLIST element attribute IDREFS default>
```

For the code sample above, to indicate that the `orders` attribute contains a list of ID references and to indicate that the `orderID` attribute contains IDs, you would enter the following attribute declarations in the DTD:

### TIP

Because an ID must be a valid XML name, it cannot begin with a number. Commonly used identifiers such as Social Security numbers must be prefaced with one or more alphabetical characters, such as SS123-45-6789.

```
<!ATTLIST customer orders IDREFS>
<!ATTLIST order orderID ID>
```

As with the `IDREF` token, all of the IDs listed in an `IDREFS` token must be found in an ID attribute located somewhere in the file; otherwise, parsers will reject the document as invalid. However, nothing in the attribute declaration defines in which attributes the referenced ID will be located. So, although the `orders` attribute defined above must reference an ID value, it is not required to find that ID value only in the `order` attribute.

In Benjamin's document, the `custID` and `orderID` attributes contain ID values, while the `orderBy` attribute contains a reference to the customer ID. You'll declare the `custID` and `orderID` attributes as `ID` data types, and the `orderBy` attribute as an `IDREF` data type.

### To declare attributes as IDs and ID references:

- 1. Within the `custID` attribute declaration, before the closing `>`, type a **space** and then type `ID`. This ensures that each `custID` value in the document is unique.
- 2. Repeat Step 1 for the `orderID` attribute declaration.
- 3. Within the `orderBy` attribute declaration, before the closing `>`, type a **space** and then type `IDREF`. This ensures that each `orderBy` value references an ID value somewhere in the document. See Figure 2-13.

Figure 2-13

Attribute IDs and IDREF added

```

<!DOCTYPE customers
[
    <!ELEMENT customers (customer+)>
    <!ELEMENT customer (name, address, phone, email?, orders)>
    <!ATTLIST customer custID ID>
    <!ELEMENT name (#PCDATA)>
    <!ATTLIST name title (Mr. | Mrs. | Ms.)>
    <!ELEMENT address (#PCDATA)>
    <!ELEMENT phone (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT orders (order+)>
    <!ELEMENT order (orderDate, items)>
    <!ATTLIST order orderID ID>
    <!ATTLIST order orderBy IDREF>
    <!ELEMENT orderDate (#PCDATA)>
    <!ELEMENT items (item+)>
    <!ELEMENT item (itemPrice, itemQty)>
    <!ATTLIST item itemNumber CDATA>
    <!ELEMENT itemPrice (#PCDATA)>
    <!ATTLIST itemPrice saleItem (Y | N)>
    <!ELEMENT itemQty (#PCDATA)>
]
>

```

- 4. Save your changes to the file.

The **NMTOKEN**, or name token, data type is used with character data whose values must meet almost all the qualifications for valid XML names. NMTOKEN data types can contain letters and numbers, as well as the underscore (`_`), hyphen (`-`), period (`.`), and colon (`:`) symbols, but not white space characters such as blank spaces or line returns. However, while an XML name can start only with a letter or an underscore, an NMTOKEN data type can begin with any valid XML character.

The limits on the NMTOKEN data type make name tokens less flexible than character data, which can contain white space characters. If Benjamin wants to make sure that an attribute value is always a valid XML name, he can use the NMTOKEN type instead of the CDATA type. For instance, he could use an NMTOKEN data type for an attribute whose value would always be a date stored in ISO date format (such as 2017-05-20). Specifying this data type would exclude obviously erroneous data, including anything with a string or a date separated by slashes (which are disallowed characters in an XML name).

When an attribute contains more than one name token, you define the attribute using the **NMTOKENS** data type and separate the name tokens in the list with blank spaces.

## Working with Attribute Defaults

The final part of an attribute declaration is the attribute default, which defines whether an attribute value is required, optional, assigned a default, or fixed. Figure 2-14 shows the entry in the attribute-list declaration for each of these four possibilities.

**Figure 2-14**

**Attribute defaults**

Attribute Default	Description
#REQUIRED	The attribute must appear with every occurrence of the element.
#IMPLIED	The attribute is optional.
" <i>default</i> "	The attribute is optional. If an attribute value is not specified, a validating XML parser will supply the <i>default</i> value.
#FIXED "default"	The attribute is optional. If an attribute value is specified, it must match the <i>default</i> value.

Figure 2-8 showed how Benjamin outlined the properties for the attributes in the orders.xml document. Based on this outline, a customer ID value is required for every customer. To indicate this in the DTD, you would add the #REQUIRED value to the attribute declaration, as follows:

```
<!ATTLIST customer custID ID #REQUIRED>
```

On the other hand, Benjamin does not always record whether a customer wants to be addressed as Mr., Mrs., or Ms., so you would add the #IMPLIED value to the title attribute to indicate that this attribute is optional. The following shows the complete attribute declaration for the title attribute:

```
<!ATTLIST name title (Mr. | Mrs. | Ms.) #IMPLIED>
```

Based on this declaration, if an XML parser encounters a name element without a title attribute, it doesn't invalidate the document but it assumes a blank value for the attribute instead.

Another attribute from Benjamin's document is the saleItem attribute, which indicates whether the itemPrice value is a sale price. The saleItem attribute is optional; however, unlike the title attribute, which gets a blank value if omitted, Benjamin wants XML parsers to assume a value of N if no value is specified for the saleItem attribute. The complete attribute declaration for the saleItem attribute is

```
<!ATTLIST itemPrice saleItem (Y | N) "N">
```

The last type of attribute default is #FIXED *default*, which fixes the attribute to the value specified by *default*. If you omit a #FIXED attribute from the corresponding element, an XML parser supplies the default value. If you include the attribute, the attribute value must equal *default* or the document is invalid.

### TIP

If you specify a default value for an attribute, omit #REQUIRED and #IMPLIED from the attribute declaration so parsers don't reject the DTD.

**REFERENCE****Specifying an Attribute Default**

- For an attribute that must appear with every occurrence of the element, insert the attribute default  
`#REQUIRED`  
within the attribute declaration.
- For an optional attribute, insert  
`#IMPLIED`
- For an optional attribute that has a value of *default* when omitted, insert  
`"default"`
- For an optional attribute that must be fixed to the value *default*, insert  
`#FIXED "default"`

Now that you've seen how to work with attribute defaults, you'll complete the attribute declarations by adding the default specifications.

**To specify attribute defaults in the orders.xml document:**

- 1. Within the `custID` attribute-list declaration, before the closing `>`, type a **space** and then type **#REQUIRED**. This indicates that this is a required attribute.
- 2. Repeat Step 1 for the `orderID`, `orderBy`, and `itemNumber` attribute-list declarations.
- 3. Within the `title` attribute-list declaration, before the closing `>`, type a **space** and then type **#IMPLIED**. This indicates that `title` is an optional attribute.
- 4. Within the `saleItem` attribute-list declaration, before the closing `>`, type a **space** and then type **"N"**. This indicates that this is the default value of the attribute if no attribute value is entered in the document. Figure 2-15 shows the final form of all of the attribute declarations in the DTD.

Figure 2-15

**Attribute defaults**

title attribute can have Mr., Mrs., or Ms. values, or the title attribute can be omitted

if the saleItem attribute is omitted, XML parsers add the attribute with the default value of N

```
<!DOCTYPE customers
[
    <!ELEMENT customers (customer+)>

    <!ELEMENT customer (name, address, phone, email?, orders)>
    <!ATTLIST customer custID ID #REQUIRED>

    <!ELEMENT name (#PCDATA)>
    <!ATTLIST name title (Mr. | Mrs. | Ms.) #IMPLIED>

    <!ELEMENT address (#PCDATA)>
    <!ELEMENT phone (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT orders (order+)>

    <!ELEMENT order (orderDate, items)>
    <!ATTLIST order orderID ID #REQUIRED> ←
    <!ATTLIST order orderBy IDREF #REQUIRED>

    <!ELEMENT orderDate (#PCDATA)>
    <!ELEMENT items (item+)>

    <!ELEMENT item (itemPrice, itemQty)>
    <!ATTLIST item itemNumber CDATA #REQUIRED> ←

    <!ELEMENT itemPrice (#PCDATA)>
    <!ATTLIST itemPrice saleItem (Y | N) "N"> ↓
    <!ELEMENT itemQty (#PCDATA)>
]
```

attributes designated as #REQUIRED must be present for the document to validate

- 5. Save your changes to the file. If you're using an XML editor, the document should once again be recognized as well formed.

## Validating an XML Document

You are ready to test whether the `orders.xml` document is valid under the rules Benjamin has specified. To test for validity, an XML parser must be able to compare the XML document with the rules established in the DTD. The web has many excellent sources for validating parsers, including websites in which you can upload an XML document for free to have it validated against an internal or external DTD. Several editors provide XML validation as well.

The following steps describe how to use Exchanger XML Editor to validate Benjamin's `orders.xml` document.

### To validate the `orders.xml` file:

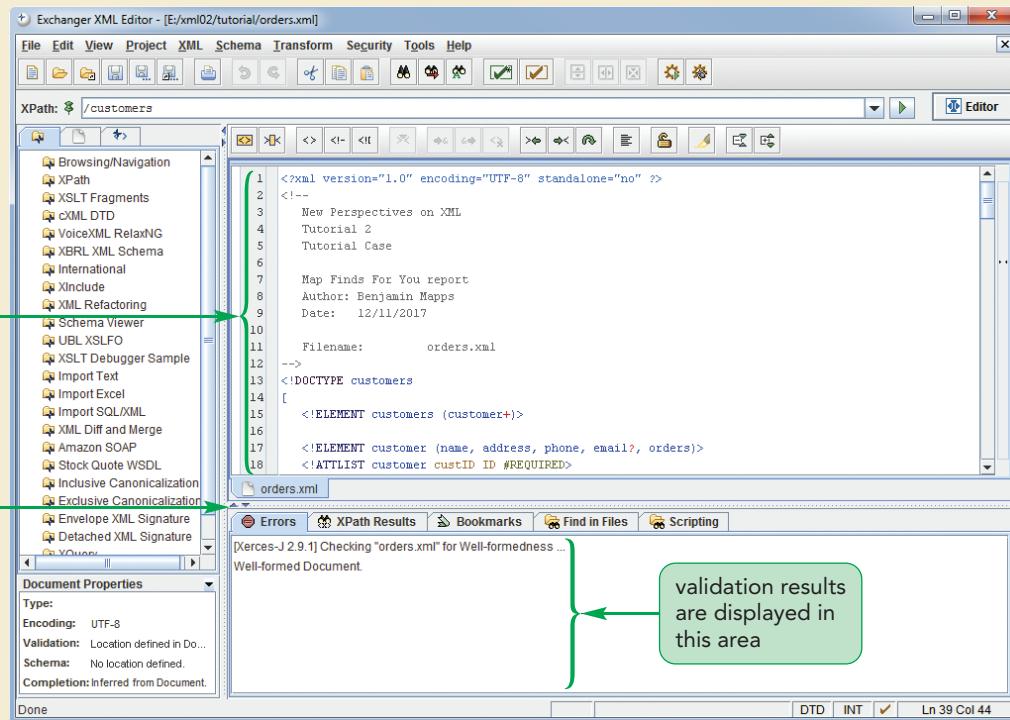
- 1. If necessary, use Exchanger XML Editor to open the `orders.xml` file.

**Trouble?** If you don't have access to Exchanger XML Editor, you can upload your document to <http://validator.w3.org>, just as you would an HTML document, to validate its content against the DTD. As long as the document is found to be valid, you can ignore any warnings generated.

2. On the menu bar, click **XML** and then on the submenu click **Validate**. As shown in Figure 2-16, the Errors tab reports that the orders.xml file is a valid document.

Figure 2-16

## Validation results in Exchanger XML Editor



**Trouble?** If the Errors tab isn't visible, click the small up arrow in the lower-left corner of the pane displaying the document code.

**Trouble?** If a validation error is reported in the Errors tab, there is an error in the DTD code. Check your DTD code against the orders.xml code shown in Figure 2-15. Your code should match exactly, including the use of uppercase and lowercase letters. Fix any discrepancies, be sure to save your changes, revalidate, and then reexamine the Errors tab for validation information.

**TIP**

You can also press F7 (Windows) or fn+F7 (Mac) to validate a document in Exchanger XML Editor.

3. If necessary, save your changes to the file.

Although the file is valid, it is a good learning experience to place a few intentional errors in your XML code to see how validation errors are discovered and reported. You'll add the following errors to your document:

- Include an element not listed in the DTD.
- Include an attribute not listed in the DTD.
- Provide a value for an attribute declared as an ID reference that does not reference any ID value in the document.

Each of these errors should cause the document to be rejected by your XML parser as invalid.

### To add intentional errors to the orders.xml file:

- 1. Return to the **orders.xml** file in your editor.
- 2. Scroll down to the first customer, John Michael, and then, directly below the **phone** element, insert a blank line and type the following new **cell** element:  
`<cell>(603) 555-1221</cell>`  
 XML parsers will flag this as an error because the element is not declared in the DTD.
- 3. Scroll down to the first **order** element, with the **orderId** value **or1089**, click just before the closing **>**, insert a **space**, and then type the following new attribute:  
`orderType="online"`  
 XML parsers will flag this as an error because the attribute is not declared in the DTD.
- 4. Within the same **order** element, change the value of the **orderBy** attribute from **cust201** to **cust210**. Because no ID attribute entered into this document has the value **cust210**, the XML parser will flag this as an error. Figure 2-17 highlights the new and revised code in the document.

Figure 2-17

### Intentional errors added to the orders.xml file

```

<customers>
  <customer custID="cust201">
    <name title="Mr.">John Michael</name>
    <address>
      <![CDATA[
        41 West Plankton Avenue
        Orlando, FL 32820
      ]]>
    </address>
    <phone>(407) 555-3476</phone>
    <cell>(603) 555-1221</cell>
    <email>jk@example.net</email>
    <orders>
      <order orderId="or1089" orderBy="cust210" orderType="online">
        <orderDate>8/11/2017</orderDate>
        <items>

```

cell element has not been declared in the DTD as an element or as a child of customer

cust210 attribute value does not match any ID listed in the document

orderType attribute has not been declared in the DTD

5. Save your changes to the file.

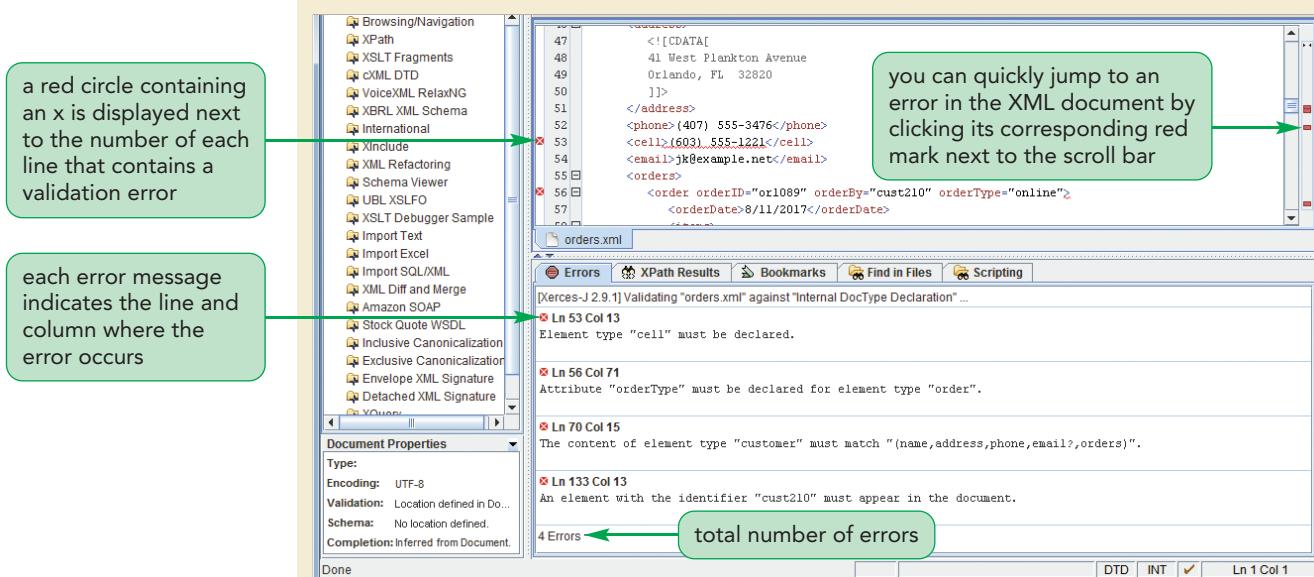
Next, you'll again validate the **orders.xml** file to test whether an XML parser catches the errors and reports them. You added three intentional errors, but it is not unusual for one error to generate more than one error message. Therefore, it is a good idea to revalidate a document after correcting each error.

### To validate the revised orders.xml file:

- 1. On the menu bar, click **XML**, and then on the submenu, click **Validate**. The parser reports two errors due to the addition of the `cell` element. First, the parser is expecting an element named `cell` to be defined. Additionally, because `cell` was not listed as a child element for the `customer` element in the DTD, another error message regarding the `customer` content type not matching is generated. Because of these errors and the additional errors generated from the other changes, the `orders.xml` file is rejected. See Figure 2-18.

Figure 2-18

### Validation errors due to an invalid document



**Trouble?** If the validation errors are not appearing, check your code against Figure 2-18 and be sure they match exactly, save the file again to ensure that the errors you added are saved, and then re-validate the document.

Note that in Exchanger XML Editor, a red circle containing an x is displayed next to the number of each line that contains a validation error. In addition, you can quickly jump to an error in the XML document by clicking its corresponding red mark next to the scroll bar.

- 2. In the `orders.xml` code, remove the `cell` element that you entered in the previous set of steps, and then save your changes to the file.
- 3. Press **F7** (Windows) or **fn+F7** (Mac) to validate the file again, and then, if necessary, scroll down the error listing. Two errors have been eliminated. Now the first error being identified involves the undeclared attribute `orderType`. See Figure 2-19.

Figure 2-19

**Reduced list of errors**

undeclared cell element removed from between these two elements

validation error due to an invalid attribute

```

<![CDATA[
41 West Plankton Avenue
Orlando, FL 32820
]]>
</address>
<phone>(407) 555-3476</phone>
<email>jk@example.net</email>
<orders>
55 <order orderId="or1089" orderBy="cust210" orderType="online">
56   <orderDate>8/11/2017</orderDate>
57   <items>
58     <item itemNumber="WM100PL">

```

**Errors**    **XPath Results**    **Bookmarks**    **Find in Files**    **Scripting**

[Xerces-J 2.9.1] Validating "orders.xml" against "Internal DocType Declaration"...

- Ln 55 Col 71 Attribute "orderType" must be declared for element type "order".
- Ln 132 Col 13 An element with the identifier "cust210" must appear in the document.

2 Errors

- 4. Remove the `orderType` attribute and its attribute value from the first `order` element, and then save your changes to the file.
- 5. Press **F7** (Windows) or **fn+F7** (Mac) to validate the file again, and then, if necessary, scroll down the error listing. Another error has been eliminated. As Figure 2-20 shows, the validator finds only one remaining error in the document: The `cust210` value does not reference any ID value found in the document.

Figure 2-20

**Final validation error**

validation error due to a missing ID value

undeclared orderType attribute removed

```

<address>
  <![CDATA[
    41 West Plankton Avenue
    Orlando, FL 32820
  ]]>
</address>
<phone>(407) 555-3476</phone>
<email>jk@example.net</email>
<orders>
55 <order orderId="or1089" orderBy="cust210">
56   <orderDate>8/11/2017</orderDate>

```

**Errors**    **XPath Results**    **Bookmarks**    **Find in Files**    **Scripting**

[Xerces-J 2.9.1] Validating "orders.xml" against "Internal DocType Declaration"...

- Ln 132 Col 13 An element with the identifier "cust210" must appear in the document.

1 Errors

- 6. Change the `orderBy` attribute value for the first `order` element from `cust210` back to **cust201**, and then save your changes to the file.
- 7. Press **F7** (Windows) or **fn+F7** (Mac) to validate the file again, and then verify that the validator reports no errors in the `orders.xml` file.



PROSKILLS

### Problem Solving: Reconciling DTDs and Namespaces

One drawback with DTDs is that they are not namespace-aware, so you cannot create a set of validation rules for elements and attributes belonging to a particular namespace. To get around this limitation, you can work with namespace prefixes, applying a validation rule to an element's qualified name. For example, if the `phone` element is placed in the `customers` namespace using the `cu` namespace prefix

```
<cu:phone>(407) 555-3476</cu:phone>
```

then the element declaration in the DTD would need to include the same element name qualification:

```
<!ELEMENT cu:phone (#PCDATA)>
```

In essence, the DTD treats a qualified name as the complete element name including the namespace prefix, colon, and local name. It doesn't recognize the namespace prefix as significant. Any namespace declarations in a document must also be included in the DTD for a document to be valid. This is usually done using a `FIXED` data type for the namespace's URI. For example, if the root element in a document declares the `customers` namespace using the attribute value

```
<cu:customers xmlns:cu="http://example.com/mapfindsforyou/ customers">
```

then the DTD should include the following attribute declaration:

```
<!ATTLIST cu:customers xmlns:cu CDATA #FIXED "http://example.com/ mapfindsforyou/customers">
```

The drawback to mixing namespaces and DTDs is that you must know the namespace prefix used in an XML document, and the DTD must be written to conform to that namespace. This makes it difficult to perform validation on a wide variety of documents that might employ any number of possible namespace prefixes. In addition, there is no way of knowing what namespace a prefix in the DTD points to because DTDs do not include a mechanism for matching a prefix to a namespace URI. It is also difficult to validate a compound document using standard vocabularies such as XHTML because you cannot easily modify the standard DTDs to accommodate the namespaces in your document.

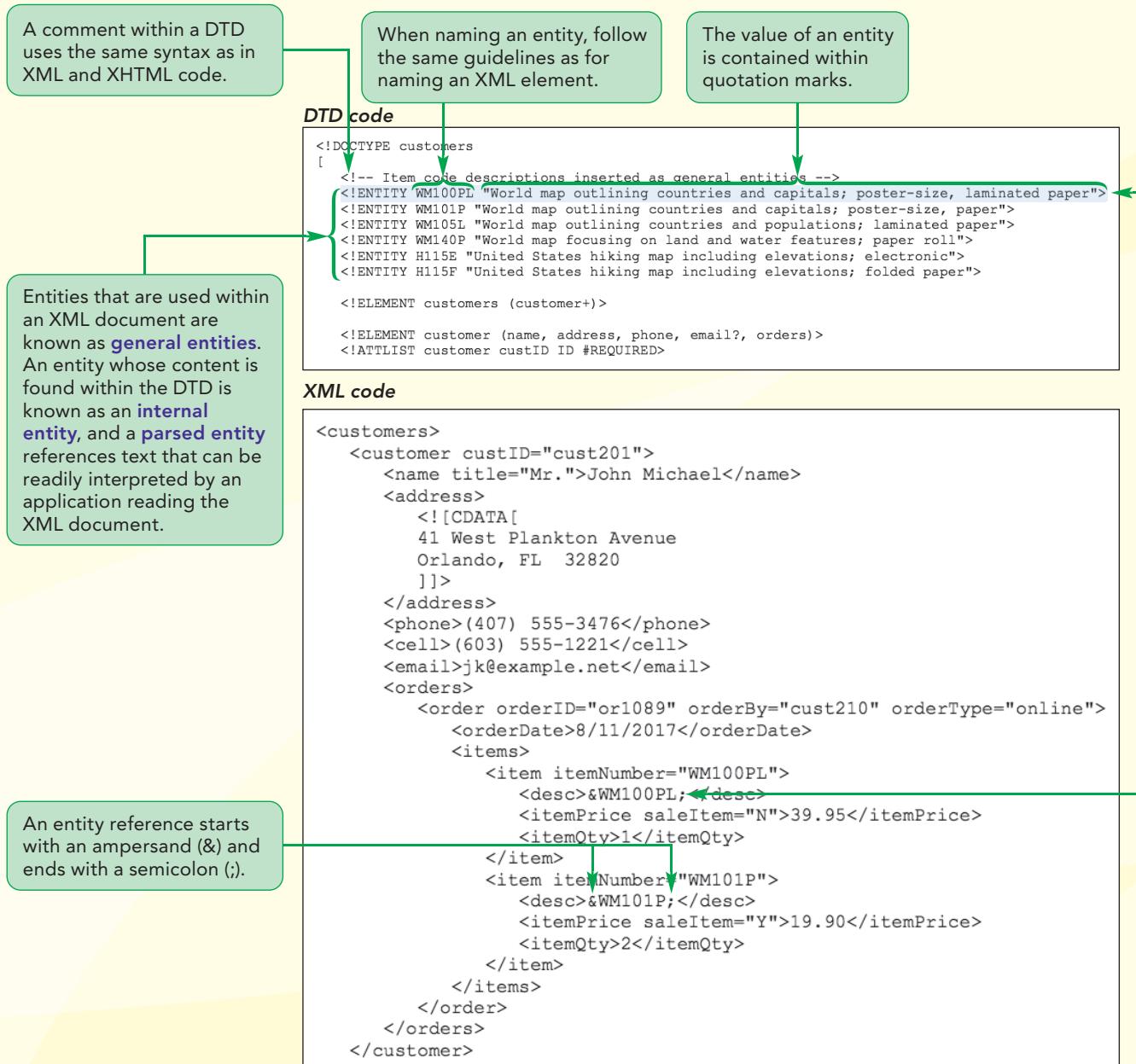
If you need to validate compound documents that employ several namespaces, a better solution is to use schemas, which are validation tools that do support namespaces.

In this session, you defined the content and structure of Benjamin's document in a DTD. In the next session, you will work with entities and nontextual content in the DTD and learn about the DTDs associated with some of the standard XML vocabularies.

**REVIEW****Session 2.2 Quick Check**

1. What attribute declaration creates an optional `title` attribute within the `book` element and specifies text string content for the `title` attribute?
2. A `play` element has a required attribute named `type`, which can have one of four possible values—`Romance`, `Tragedy`, `History`, and `Comedy`. Provide the appropriate attribute declaration for the `type` attribute.
3. What is the main difference between an attribute with the CDATA type and one with the NMTOKEN type?
4. A `book` element has a required ID attribute named `ISBN`. Provide the appropriate attribute declaration for the `ISBN` attribute.
5. An `author` element has an optional attribute named `booksBy`, which contains a white space-separated list of ISBNs for the books the author has written. If `ISBN` is an ID attribute for another element in the document, what declaration would you use for the `booksBy` attribute?
6. A `book` element has an optional attribute named `inStock` that can have the value `yes` or `no`. The default value is `yes`. What is the declaration for the `inStock` attribute?

## Session 2.3 Visual Overview:



# Entities and Comments in a DTD

An entity reference is replaced by its value when an XML document is parsed.

## Parsed XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- New Perspectives on XML Tutorial 2 Tutorial Case Map Finds For You report Author: Benjamin Mapps Date:
12/11/2017 Filename: orders.xml -->
<!DOCTYPE customers>
- <customers>
  - <customer custID="cust201">
    <name title="Mr.">John Michael</name>
    - <address>
      <![CDATA[ 41 West Plankton Avenue Orlando, FL 32820 ]]>
    </address>
    <phone>(407) 555-3476</phone>
    <email>jk@example.net</email>
  - <orders>
    - <order orderBy="cust201" orderID="or1089">
      <orderDate>8/11/2017</orderDate>
      - <items>
        - <item itemNumber="WM100PL">
          <desc>World map outlining countries and capitals; poster-size, laminated paper</desc>
          <itemPrice saleItem="N">39.95</itemPrice>
          <itemQty>1</itemQty>
        </item>
        - <item itemNumber="WM101P">
          <desc>World map outlining countries and capitals; poster-size, paper</desc>
          <itemPrice saleItem="Y">19.90</itemPrice>
          <itemQty>2</itemQty>
        </item>
      </items>
    </order>
  </orders>
</customer>
```

## Introducing Entities

In the orders.xml document, Benjamin inserted item numbers for the different items ordered by customers. For example, the first customer in the file, Mr. John Michael, ordered two items with the item codes WM100PL and WM101P. Each of these item numbers is associated with a longer text description of the product. Figure 2-21 shows the item code and item description for each of the items in the orders.xml file.

Figure 2-21

Item codes and descriptions

Item Codes	Description
WM100PL	World map outlining countries and capitals; poster-size, laminated paper
WM101P	World map outlining countries and capitals; poster-size, paper
WM105L	World map outlining countries and populations; laminated paper
WM140P	World map focusing on land and water features; paper roll
H115E	United States hiking map including elevations; electronic
H115F	United States hiking map including elevations; folded paper

Benjamin wants some way to include the longer text descriptions of these items in his document without having to enter them into each order, which might result in some typographical errors. You can do this with entities. You have already worked with entities to insert special character strings into an XML document. XML supports the following five built-in entities:

- &#38; for the & character
- &lt; for the < character
- &gt; for the > character
- &apos; for the ' character
- &quot; for the " character

When an XML parser encounters one of these entities, it can display the corresponding character symbol. With a DTD, you can create a customized set of entities corresponding to text strings such as Benjamin's product descriptions, files, or nontextual content that you want referenced by the XML document. When an XML parser encounters one of your customized entities, it will also be able to display the corresponding character text.

## Working with General Entities

### TIP

For a long text string that will be repeated throughout an XML document, avoid data entry errors by placing the text string in its own entity.

To create a customized entity, you add it to the document's DTD. An entity is classified based on three factors—where it will be applied, where its content is located, and what type of content it references. Entities that are used within an XML document are known as general entities. Another type of entity, a **parameter entity**, is used within a DTD. You will work with parameter entities later in this session.

Entities can reference content found either in an external file or within the DTD itself. An entity that references content found in an external file is called an **external entity**, whereas an entity whose content is found within the DTD is known as an internal entity.

Finally, the content referenced by an entity can be either parsed or unparsed. A parsed entity references text that can be readily interpreted, or parsed, by an application reading the XML document. Parsed entities are used when text is repeated often, and can reference characters, words, phrases, paragraphs, or entire documents. The only requirement is that the text be well formed. An entity that references content that either is nontextual or cannot be interpreted by an XML parser is an **unparsed entity**. One example of an unparsed entity is an entity that references a graphic image file.

Different types of entities are declared slightly differently based on the above three factors. For Benjamin's product codes, you'll declare general parsed internal entities.

## Creating Parsed Entities

To create a parsed internal entity, you add the entity declaration

```
<!ENTITY entity "value">
```

to the DTD, where *entity* is the name assigned to the entity and *value* is the text string associated with the entity. The entity name follows the same rules that apply to all XML names: It can have no blank spaces and must begin with either a letter or an underscore. The entity value itself must be well-formed XML text. This can be a simple text string or it can be well-formed XML code. For example, to store the product description for the product with the item code WM100PL, you would add the following entity to the document's DTD:

```
<!ENTITY WM100PL "World map outlining countries and capitals;
poster-size, laminated paper">
```

### TIP

Including markup tags in an entity value lets you create a section of XML code and content, and insert it once or multiple times into a document.

You can add markup tags to any entity declaration by including them in the entity value, as follows:

```
<!ENTITY WM100PL "<desc>World map outlining countries and capitals;
poster-size, laminated paper</desc>">
```

Any text is allowed to be used for an entity's value as long as it corresponds to well-formed XML. Thus, a document containing the following entity declaration would not be well formed because the declaration lacks the closing *</desc>* tag in the entity's value:

### Not well-formed code:

```
<!ENTITY WM100PL "<desc>World map outlining countries and capitals;
poster-size, laminated paper">
```

Entity values must be well formed without reference to other entities or document content. You couldn't, for example, place the closing *</desc>* tag in another entity declaration or within the XML document itself. Although tags within entity references are allowed, this is not a recommended practice.

For longer text strings that would not easily fit within the value of an entity declaration, you can place the content in an external file. To create a parsed entity that references content from an external file using a system identifier, such as a filename or URI, you use the declaration

```
<!ENTITY entity SYSTEM "uri">
```

where *entity* is again the entity's name, *SYSTEM* indicates the content is located in an external file, and *uri* is the URI of the external file containing the entity's content. For instance, the following entity declaration references the content of the *description.xml* file:

```
<!ENTITY WM100PL SYSTEM "description.xml">
```

The *description.xml* file must contain well-formed XML content. However, it should not contain an XML declaration. Because an XML document can contain only one XML declaration, placing a second one in a document via an external entity results in an error.

An external entity can also reference content from an external file using a public identifier with the declaration

```
<!ENTITY entity PUBLIC "id" "uri">
```

where *PUBLIC* indicates the content is located in a public location, *id* is the public identifier, and *uri* is the system location of the external file (included in case an XML

parser doesn't recognize the public identifier). An entity declaration referencing a public location might look like the following:

```
<!ENTITY WM100PL PUBLIC "-//MFY// WM100PL INFO" "description.xml">
```

In this case, the public identifier `-//MFY// WM100PL INFO` is used by an XML parser to load the external content corresponding to the `WM100PL` entity. If that identifier is not recognized, the parser falls back to the system location of the `description.xml` file. In this way, external entities behave like DOCTYPEs that reference external DTDs from either public or system locations.

## REFERENCE

### *Declaring and Referencing Parsed Entities*

- To declare a parsed internal entity, use the declaration

```
<!ENTITY entity "value">
```

where `entity` is the entity's name and `value` is the entity's value.

- To declare a parsed external entity from a system location, use the declaration

```
<!ENTITY entity SYSTEM "uri">
```

where `uri` is the URI of the external file containing the entity value.

- To declare a parsed external entity from a public location, use the declaration

```
<!ENTITY entity PUBLIC "id" "uri">
```

where `id` is the public identifier for the external file.

- To reference a general entity within an XML document, enter

`&entity;`

where `entity` is the entity name declared in the DTD associated with the XML document.

## Referencing a General Entity

After a general entity is declared in a DTD, it can be referenced anywhere within the body of the XML document. The syntax for referencing a general entity is the same as for referencing one of the five built-in XML entities, namely

`&entity;`

where `entity` is the entity's name as declared in the DTD. For example, if the `WM100PL` entity is declared in the DTD as

```
<!ENTITY WM100PL "World map outlining countries and capitals;
poster-size, laminated paper">
```

you could reference the entity's value in the XML document within a `desc` element as follows:

```
<desc>&WM100PL;</desc>
```

Any XML parser encountering this entity reference would be able to expand the entity into its referenced value, resulting in the following parsed code:

```
<desc>World map outlining countries and capitals; poster-size,
laminated paper</desc>
```

The fact that the entity's value is expanded into the code of the XML document is one reason why entity values must correspond to well-formed XML code. Because of the way entities are parsed, you cannot include the & symbol as part of an entity's value.

XML parsers interpret the & symbol as a reference to another entity and attempt to resolve the reference. If you need to include the & symbol, you should use the built-in entity reference &amp;. You also cannot use the % symbol in an entity's value because, as you'll learn later in this session, this is the symbol used for inserting parameter entities.

At the time of this writing, none of the major browsers support the use of external entities in combination with DTDs. These browsers use a built-in XML parser that does not support resolution of external entities. The reason is that if an entity declaration is placed in a file on a remote web server, the XML parser must establish a TCP/IP connection with the remote file, which might not always be possible. The major browsers consider such a connection to be a potential security risk. Thus, to ensure that an XML document can be properly read and rendered, browsers require entities to be part of the internal DTD.

Next, you'll declare parsed entities for all the product codes in the orders.xml document. You'll place the declarations within the internal DTD. You'll also add a declaration to the DTD for the desc element, which will contain the entities within the document code.

### To create the entity declarations in an internal DTD:

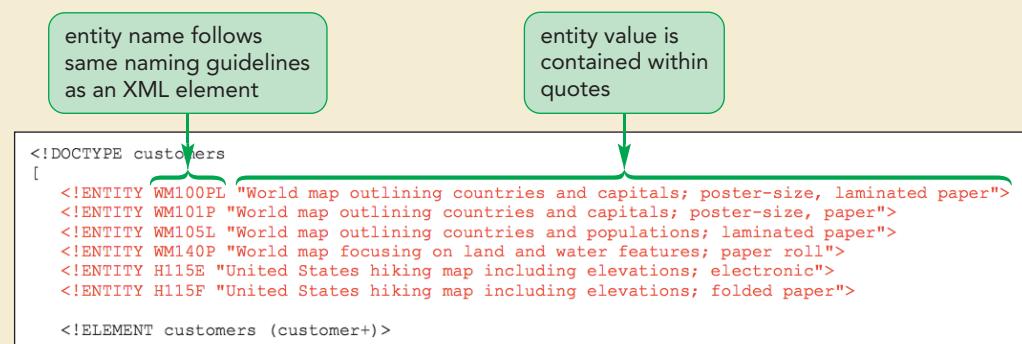
- 1. If you took a break after the previous session, make sure the **orders.xml** file is open in your text editor.
- 2. Add the following entity declarations below the opening square bracket in the DOCTYPE declaration statement:

```
<!ENTITY WM100PL "World map outlining countries and capitals;
    poster-size, laminated paper">
<!ENTITY WM101P "World map outlining countries and capitals;
    poster-size, paper">
<!ENTITY WM105L "World map outlining countries and populations;
    laminated paper">
<!ENTITY WM140P "World map focusing on land and water features;
    paper roll">
<!ENTITY H115E "United States hiking map including elevations;
    electronic">
<!ENTITY H115F "United States hiking map including elevations;
    folded paper">
```

Figure 2-22 shows the updated DOCTYPE.

**Figure 2-22**

**Creating general entities**



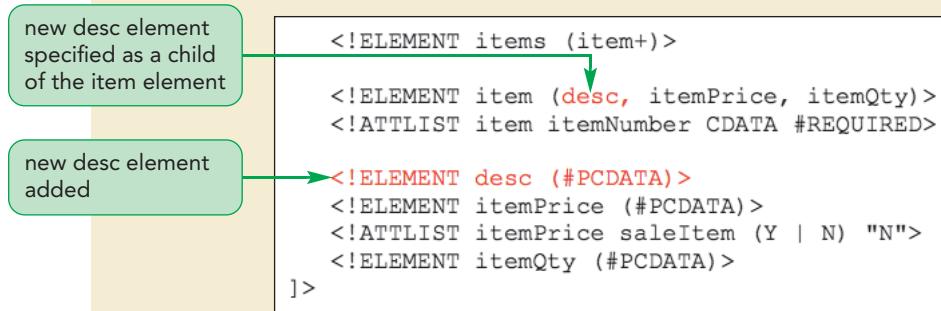
- 3. Scroll down to the element definition for the item element, place the insertion point after the opening parenthesis for the list of child elements and before the itemPrice element name, and then type desc followed by a comma and a space.

- 4. Insert a blank line before the `itemPrice` element definition, and then type the following new element definition on the blank line you just created:

```
<!ELEMENT desc (#PCDATA)>
```

Figure 2-23 shows the updated code in the DTD.

**Figure 2-23 Adding a new element to the DTD**



- 5. Save the `orders.xml` file.

Now that you've defined the entities, you'll insert entity references to Benjamin's item description codes in the body of the XML document.

#### To add entity references:

- 1. Scroll down to the `customer` element for the first customer, John Michael, and then directly below the opening tag for the first `item` element, enter the following new `desc` element:

```
<desc>&WM100PL;</desc>
```

Notice that the entity reference is the value of the `itemNumber` attribute for the preceding `item` element, with & at the start and ; at the end.

- 2. Directly below the opening tag for the second `item` element for John Michael, add the following new `desc` element:

```
<desc>&WM101P;</desc>
```

Figure 2-24 shows the revised code for John Michael's orders.

Figure 2-24

## Revised order information for John Michael

```

<customer custID="cust201">
  <name title="Mr.">John Michael</name>
  <address>
    <![CDATA[
      41 West Plankton Avenue
      Orlando, FL 32820
    ]]>
  </address>
  <phone>(407) 555-3476</phone>
  <cell>(603) 555-1221</cell>
  <email>jk@example.net</email>
  <orders>
    <order orderID="or1089" orderBy="cust210" orderType="online">
      <orderDate>8/11/2017</orderDate>
      <items>
        <item itemNumber="WM100PL">
          <desc>&WM100PL;</desc>
          <itemPrice saleItem="N">39.95</itemPrice>
          <itemQty>1</itemQty>
        </item>
        <item itemNumber="WM101P">
          <desc>&WM101P;</desc>
          <itemPrice saleItem="Y">19.90</itemPrice>
          <itemQty>2</itemQty>
        </item>
      </items>
    </order>
  </orders>
</customer>

```

desc element containing the general entity reference for the item description

general entity reference begins with an ampersand and ends with a semicolon

Be sure to include an ampersand ( & ) at the start of each entity reference and a semicolon ( ; ) at the end of each one; otherwise, parsers will not correctly interpret them as entity references.

3. Insert a desc child element within each of the four item elements for Dean Abernath, each containing an ampersand ( & ) followed by the value of the itemNumber attribute for the preceding item element and then a semicolon ( ; ), as shown in Figure 2-25.

Figure 2-25

## Revised order information for Dean Abernath

```

<email>dabernath@example.net</email>
<orders>
    <order orderID="or1021" orderBy="cust202">
        <orderDate>8/1/2017</orderDate>
        <items>
            <item itemNumber="WM100PL">
                <desc>&WM100PL;</desc>
                <itemPrice>29.95</itemPrice>
                <itemQty>1</itemQty>
            </item>
            <item itemNumber="WM105L">
                <desc>&WM105L;</desc>
                <itemPrice>19.95</itemPrice>
                <itemQty>1</itemQty>
            </item>
        </items>
    </order>
    <order orderID="or1122" orderBy="cust202">
        <orderDate>10/1/2017</orderDate>
        <items>
            <item itemNumber="H115E">
                <desc>&H115E;</desc>
                <itemPrice saleItem="Y">24.90</itemPrice>
                <itemQty>2</itemQty>
            </item>
            <item itemNumber="H115F">
                <desc>&H115F;</desc>
                <itemPrice saleItem="N">14.95</itemPrice>
                <itemQty>1</itemQty>
            </item>
        </items>
    </order>
</orders>
```

- 4. Repeat Step 3 for the one remaining `item` element for the last customer, Riverfront High School. Figure 2-26 shows the revised code for the Riverfront High School order.

Figure 2-26

## Revised order information for Riverfront High School

```

<phone>(715) 555-4022</phone>
<orders>
    <order orderID="or1120" orderBy="cust203">
        <orderDate>9/15/2017</orderDate>
        <items>
            <item itemNumber="WM140P">
                <desc>&WM140P;</desc>
                <itemPrice>78.90</itemPrice>
                <itemQty>2</itemQty>
            </item>
        </items>
    </order>
</orders>
```

- 5. Save the `orders.xml` file.

Now that you've added the entity references to Benjamin's document, you can verify that their values are resolved in your web browser.

6. Open **orders.xml** in a web browser. When viewed in a browser that uses an outline format for XML files, the values of all seven product codes are displayed. Figure 2-27 shows the document in Internet Explorer.

**Figure 2-27****The orders.xml file displayed in a browser**

parser translation of the &WM100PL; entity reference you entered in the desc element

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- New Perspectives on XML Tutorial 2 Tutorial Case Map Finds For You report Author: Benjamin Mapps Date:
12/11/2017 Filename: orders.xml -->
<!DOCTYPE customers>
- <customers>
  - <customer custID="cust201">
    <name title="Mr.">John Michael</name>
    - <address>
      <![CDATA[ 41 West Plankton Avenue Orlando, FL 32820 ]]>
    </address>
    <phone>(407) 555-3476</phone>
    <email>jk@example.net</email>
  - <orders>
    - <order orderBy="cust201" orderID="or1089">
      <orderDate>8/11/2017</orderDate>
      - <items>
        - <item itemNumber="WM100PL">
          <desc>World map outlining countries and capitals; poster-size, laminated paper</desc>
          <itemPrice saleItem="N">39.95</itemPrice>
          <itemQty>1</itemQty>
        </item>
        - <item itemNumber="WM101P">
          <desc>World map outlining countries and capitals; poster-size, paper</desc>
          <itemPrice saleItem="Y">19.90</itemPrice>
          <itemQty>2</itemQty>
        </item>
      </items>
    </order>
  </orders>
</customer>
```

**Trouble?** If your browser displays a warning message about the content of your file, click the necessary button(s) to allow the content to be parsed and displayed. Some browsers treat XML files with suspicion for computer security reasons; but because you created this file, you don't need to be concerned about opening it.

So far, you've placed all your entity declarations internally within the `orders.xml` document. However, you can also externally link declarations rather than defining them internally. There are two methods for linking to external declarations—revising the DOCTYPE to link to an external DTD file and using a parameter entity.

## Working with Parameter Entities

Just as you use a general entity when you want to insert content into an XML document, you use a **parameter entity** when you want to insert content into the DTD itself. You can use parameter entities to break a DTD into smaller chunks, or **modules**, that are placed in different files. Imagine a team of programmers working on a DTD for a large XML vocabulary such as XHTML, containing hundreds of elements and attributes. Rather than placing all the declarations within a single file, individual programmers could work on sections suited to their expertise. Then a project coordinator could use parameter entities to reference the different sections in the main DTD. Parameter entities also enable XML programmers to reuse large blocks of DTD code, which saves them from retyping the same code multiple times.

The declaration to create a parameter entity is similar to the declaration for a general entity, with the syntax

```
<!ENTITY % entity "value">
```

where `entity` is the name of the parameter entity and `value` is the text referenced by the parameter entity. Like general entities, parameter entities can also reference external

content in either system or public locations. The declarations for external parameter entities are

```
<!ENTITY % entity SYSTEM "uri">
```

and

```
<!ENTITY % entity PUBLIC "id" "uri">
```

where *id* is a public identifier for the parameter entity and *uri* is the location of the external file containing DTD content.

For example, the following code shows an internal parameter entity for a collection of elements and attributes:

```
<!ENTITY % books
  "<!ELEMENT Book (Title, Author)>
   <!ATTLIST Book Pages CDATA #REQUIRED>
   <!ELEMENT Title (#PCDATA)>
   <!ELEMENT Author (#PCDATA)>">
```

If you instead placed the elements and attributes referenced in the previous code in an external DTD file named *books.dtd*, you could use an external parameter entity to access the content of that document as follows:

```
<!ENTITY % books SYSTEM "books.dtd">
```

After a parameter has been declared, you can reference it within the DTD using the statement

`%entity;`

**TIP**

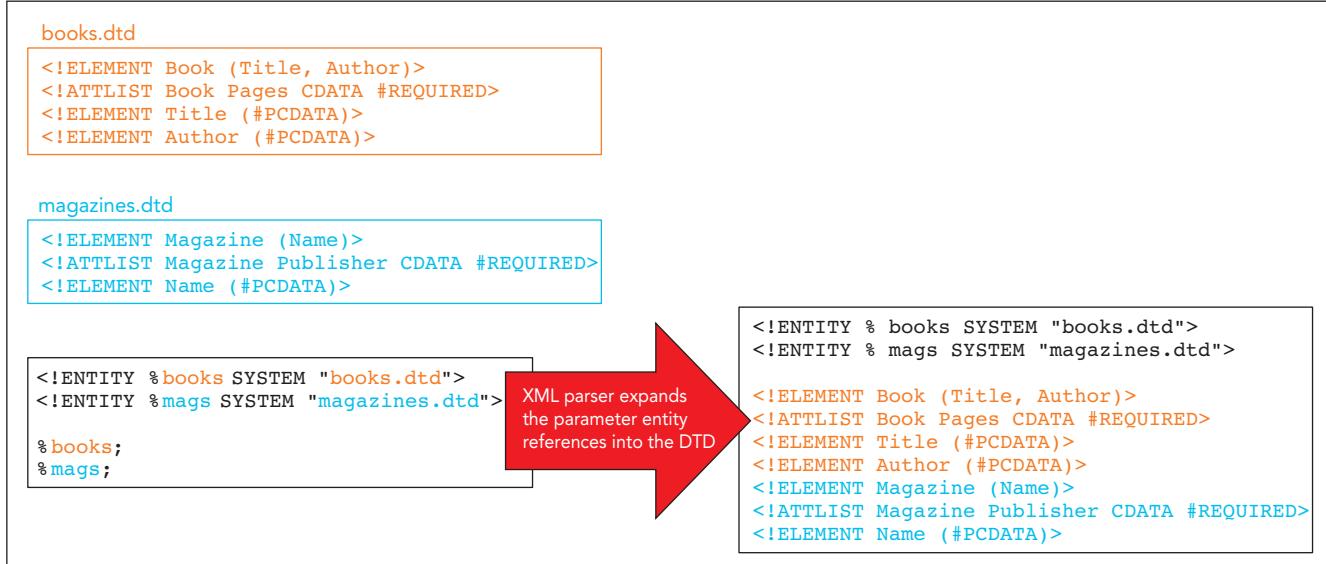
Note that when declaring a parameter entity, you include a space after the %; but when referencing a parameter entity, there is no space between the % and the entity name.

where *entity* is the name assigned to the parameter entity. Parameter entity references can be placed only where a declaration would regularly occur, such as within an internal or external DTD. You *cannot* insert a parameter entity reference within the element content of an XML document. For example, to reference the *books* parameter entity described above, you would enter the following line into the DTD:

`%books;`

Figure 2-28 shows how parameter entities can be used to combine DTDs from multiple files into a single (virtual) DTD.

Figure 2-28 Creating a combined DTD using parameter entities



In the figure, two parameter entities—`%books;` and `%mags;`—are used to combine the contents of books.dtd and magazines.dtd into a master document.

**REFERENCE**

### Declaring and Referencing Parameter Entities

- To declare an internal parameter entity, add the line  
`<!ENTITY % entity "value">`  
to the DTD, where `entity` is the entity name and `value` is the entity value.
- To declare an external parameter entity for a system location, use  
`<!ENTITY % entity SYSTEM "uri">`  
where `uri` is the location of the external file containing DTD content.
- To declare an external parameter entity for a public location, use  
`<!ENTITY % entity PUBLIC "id" "uri">`  
where `id` is the public identifier.
- To reference a parameter entity, add the statement  
`%entity;`  
to the DTD, where `entity` is the name of the parameter entity.

## Inserting Comments into a DTD

Benjamin is pleased with your work on creating the DTD for the sample XML document. However, he is concerned that the code in the DTD might be confusing to other programmers. He suggests that you add a comment. Comments in a DTD follow the same syntax as comments in XML. The specific form of a DTD comment is

```
<!-- comment -->
```

where `comment` is the text of the DTD comment. White space is ignored within a comment, so you can spread comment text over several lines without affecting DTD code.

**PROSKILLS**

### Teamwork: Documenting Shared Code with Comments

When you're working on a large project with other developers to create a DTD, including comments in your code is important to help other team members quickly understand the code you've written. You can use comments to indicate changes you've made to existing code, to flag code that may need future changes, and simply to document what you've done and when. Although a DTD doesn't require comments, using them to document your work can make you a more valuable team member and increase your group's efficiency.

You'll add a comment now to the orders.xml DTD to summarize its content.

#### To add a comment to the DTD:

1. Return to the `orders.xml` file in your text editor.

- Within the DTD, after the opening bracket and above the first general entity (for the WM100PL item code), insert a blank line, and then enter the following comment:

```
<!-- Item code descriptions inserted as general entities -->
```

Figure 2-29 shows the final form of the DTD for the orders.xml file.

**Figure 2-29**

### DTD comment added

```
<!DOCTYPE customers
[
    <!-- Item code descriptions inserted as general entities -->
    <!ENTITY WM100PL "World map outlining countries and capitals; poster-size, laminated paper">
    <!ENTITY WM101P "World map outlining countries and capitals; poster-size, paper">
    <!ENTITY WM105L "World map outlining countries and populations; laminated paper">
    <!ENTITY WM140P "World map focusing on land and water features; paper roll">
    <!ENTITY H115E "United States hiking map including elevations; electronic">
    <!ENTITY H115F "United States hiking map including elevations; folded paper">

    <!ELEMENT customers (customer+)>
```

- Save your changes to the **orders.xml** file, and then close the file.

## Creating Conditional Sections

When you're creating a new DTD, it can be useful to try out different combinations of declarations. You can do this by using a **conditional section**, which is a section of the DTD that is processed only in certain situations. The syntax for creating a conditional section is

```
<![keyword[
    declarations
]]>
```

where *keyword* is either **INCLUDE** (for a section of declarations that you want parsers to interpret) or **IGNORE** (for the declarations that you want parsers to pass over).

For example, the following code creates two sections of declarations—one for the **Magazine** element and its child elements, and another for the **Book** element and its child elements—and instructs parsers to ignore the **Magazine**-related elements and interpret the **Book**-related elements:

```
<![ IGNORE[
    <!ELEMENT Magazine (Name)>
    <!ATTLIST Magazine Publisher CDATA #REQUIRED>
    <!ELEMENT Name (#PCDATA)>
]]>

<![ INCLUDE[
    <!ELEMENT Book (Title, Author)>
    <!ATTLIST Book Pages CDATA #REQUIRED>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
]]>
```

An XML parser processing this DTD would run the declarations related to the **Book** element, but would ignore the declarations related to the **Magazine** element. As you experiment with a DTD's structure, you can enable a section by changing its keyword from **IGNORE** to **INCLUDE**.

One effective way of creating `IGNORE` sections is to create a parameter entity that defines whether those sections should be included, and to use the value of the entity as the keyword for the conditional sections. For example, the following `UseFullDTD` entity has a value of `IGNORE`, which causes the conditional section that follows it to be ignored by XML parsers:

```
<!ENTITY % UseFullDTD "IGNORE">

<![ %UseFullDTD; [
  <!ELEMENT Magazine (Name)>
  <!ATTLIST Magazine Publisher CDATA #REQUIRED>
  <!ELEMENT Name (#PCDATA)>
]]>
```

By changing the value of `UseFullDTD` from `IGNORE` to `INCLUDE`, you can add any conditional section that uses this entity reference to the document's DTD. This enables you to switch multiple sections in the DTD on and off by editing a single line in the file, which is most useful when several conditional sections are scattered throughout a long DTD. Rather than locating and changing each conditional section, you can switch the sections on and off by changing the parameter entity's value.

Conditional sections can be applied only to external DTDs. Both sets of code samples above that illustrate conditional sections would need to be located in external files, rather than in the internal subset of a DTD. Although they may be useful in other contexts, you cannot apply them to the DTD in Benjamin's document because it uses only an internal DTD.

## Working with Unparsed Data

So far in this session, you've created entities for character data. For a DTD to validate either binary data, such as images or video clips, or character data that is not well formed, you need to work with unparsed entities. Because an XML parser cannot work with these types of data directly, a DTD must include instructions for how to treat an unparsed entity.

The first step is to declare a **notation**, which identifies the data type of the unparsed data. A notation must supply a name for the data type and provide clues about how applications should handle the data. Notations must reference external content (because that content must contain nontextual data that is, by definition, not well formed) and you must specify an external location. One option is to use a system location, which you specify with the code

```
<!NOTATION notation SYSTEM "uri">
```

where `notation` is the notation's name and `uri` is a system location that gives the XML parser clues about how the data should be handled. The other option is to specify a public location, using the declaration

```
<!NOTATION notation PUBLIC "id" "uri">
```

where `id` is a public identifier recognized by XML parsers. The URI for the resource can either refer to a program that can work with the unparsed data or specify the actual data type. For example, if Benjamin wanted to include references in the `orders.xml` document to graphic image files stored in the PNG format, he could enter the following notation in the document's DTD:

```
<!NOTATION png SYSTEM "paint.exe">
```

Because an XML parser doesn't know how to handle graphic data, this notation associates the `paint.exe` program with the `png` data type. If you don't want to specify

a particular program, you could instead indicate the data type using its MIME type value with the following notation:

```
<!NOTATION png SYSTEM "image/png">
```

In this case, an XML parser associates the `png` notation with the `image/png` data type as long as the operating system already knows how to handle PNG files. After a notation is declared, you can create an unparsed entity that references specific items that use that notation. The syntax to declare an unparsed entity is

```
<!ENTITY entity SYSTEM "uri" NDATA notation>
```

where `entity` is the name of the entity referencing the notation, `uri` is the URI of the unparsed data, and `notation` is the name of the notation that defines the data type for the XML parser. Again, you can also provide a public location for the unparsed data if an XML parser supports it, using the following form:

```
<!ENTITY entity PUBLIC "id" "uri" NDATA notation>
```

For example, the following declaration creates an unparsed entity named `WM100PLIMG` that references the graphic image file `WM100PL.png`:

```
<!ENTITY WM100PLIMG SYSTEM "WM100PL.png" NDATA png>
```

This declaration references the `png` notation created above to provide the data type.

After you create an entity to reference unparsed data, that entity can be associated with attribute values by using the `ENTITY` data type in each attribute declaration. If Benjamin wanted to add an `image` attribute to every `item` element in the `orders.xml` document, he could insert the following attribute declaration in the DTD:

```
<!ATTLIST item image ENTITY #REQUIRED>
```

With this declaration added, Benjamin could then add the `image` attribute to the XML document, using the `WM100PLIMG` entity as the attribute's value, as follows:

```
<item image="&WM100PLIMG;">
```

### TIP

As an alternative to notations, you can place a URL that lists a resource for nontextual content in an element or attribute, and then allow your application to work with that element or attribute value directly.

It's important to understand precisely what this code does and does not accomplish. It tells XML parsers what kind of data is represented by the `WM100PLIMG` entity, and it provides clues about how to interpret the data stored in the `WM100PL.png` file—but it does not tell parsers anything else. Whether an application reading the XML document opens another application to display the image file depends solely on the program itself. Remember that the purpose of XML is to create structured documents, but not necessarily to tell programs how to render the data in a document. If a validating XML parser reads the `<item>` tag described above, it probably wouldn't try to read the graphic image file, but it might check to see whether the file is in the expected location. By doing so, the parser would confirm that the document is complete in its content and in all its references to unparsed data.

Current web browsers do not support mechanisms for validating and rendering unparsed data declared in the DTDs of XML documents, so you will not add this feature to the `orders.xml` file.

## REFERENCE

### Declaring an Unparsed Entity

- To declare an unparsed entity, first declare a notation for the data type used in the entity, using the syntax

```
<!NOTATION notation SYSTEM "uri">
```

where *notation* is the name of the notation and *uri* is a system location that defines the data type or a program that can work with the data type.

- To specify a public location for the notation, use the declaration

```
<!NOTATION notation PUBLIC "id" "uri">
```

where *id* is a public identifier for the data type associated with the notation.

- To associate a notation with an unparsed entity, use the declaration

```
<!ENTITY entity SYSTEM "uri" NDATA notation>
```

where *entity* is the name of the entity, *uri* is the system location of a file containing the unparsed data, and *notation* is the name of the notation that defines the data type.

- For a public location of the unparsed entity, use the following declaration:

```
<!ENTITY entity PUBLIC "id" "uri" NDATA notation>
```

## Validating Standard Vocabularies

All of your work in this tutorial involved the custom XML vocabulary developed by Benjamin for orders submitted to Map Finds For You. Most of the standard XML vocabularies in popular use are associated with existing DTDs. To validate a document used with a standard vocabulary, you usually must access an external DTD located on a web server or rely upon a DTD built into your XML parser. Figure 2-30 lists the DOCTYPEs for some popular XML vocabularies.

**Figure 2-30**

**DOCTYPEs for standard vocabularies**

Vocabulary	DOCTYPE
XHTML 1.0 Strict	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
XHTML 1.0 Transitional	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
XHTML 1.1	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
MathML 1.01	<!DOCTYPE math SYSTEM "http://www.w3.org/Math/DTD/mathml1/mathml.dtd">
MathML 2.0	<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN" "http://www.w3.org/Math/DTD/mathml2/mathml2.dtd">
SVG 1.1 Basic	<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1 Basic//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-basic.dtd">
SVG 1.1 Full	<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
SMIL 1.0	<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 1.0//EN" "http://www.w3.org/TR/REC-smil/SMIL10.dtd">
SMIL 2.0	<!DOCTYPE SMIL PUBLIC "-//W3C//DTD SMIL 2.0//EN" "http://www.w3.org/TR/REC-smil/2000/SMIL20.dtd">
VoiceXML 2.1	<!DOCTYPE vxml PUBLIC "-//W3C//DTD VOICEXML 2.1//EN" "http://www.w3.org/TR/voicesml21/vxml.dtd">

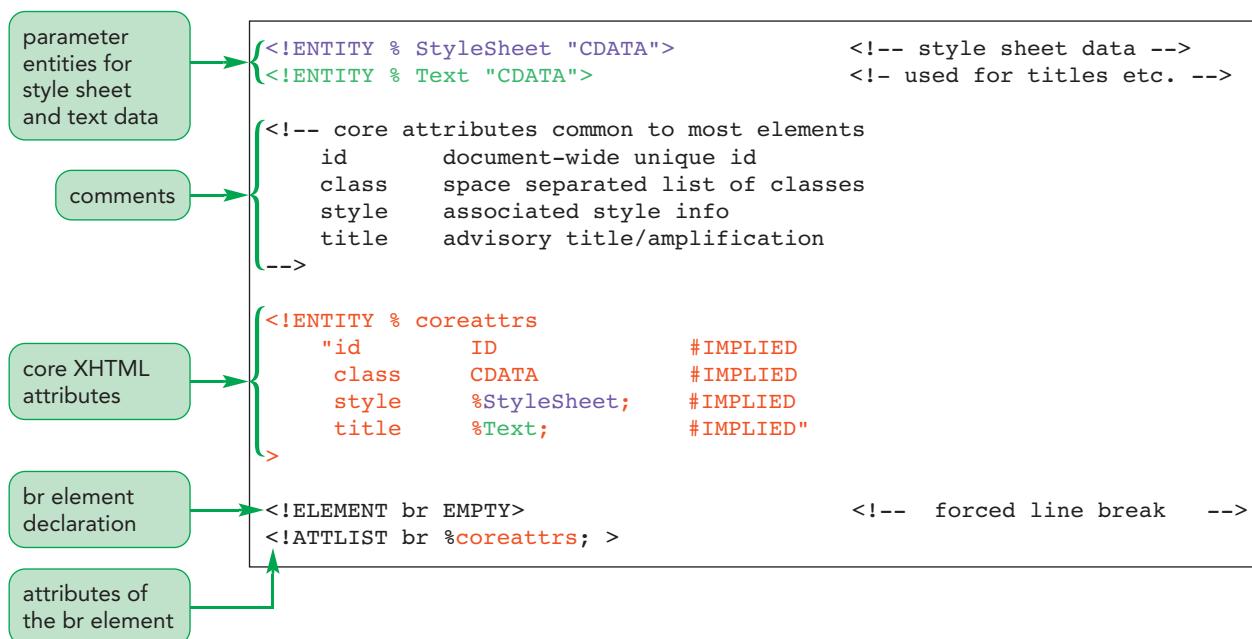
For example, to validate an XHTML document against the XHTML 1.0 Strict standard, you would add the following code to the document header:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

The W3C provides an online validator at <http://validator.w3.org> that you can use to validate HTML, XHTML, MathML, SVG, and other XML vocabularies. The validator works with files placed on the web or uploaded via a web form.

The DTDs of most standard vocabularies are available online for inspection. Studying the DTDs of other XML vocabularies is a great way to learn how to design your own. Figure 2-31 shows the part of the DTD for XHTML 1.0 that sets the syntax rules for the **br** (line break) element.

**Figure 2-31** **XHTML 1.0 Strict DTD for the br element**



This DTD includes substantial use of parameter entities to allow the same set of attributes to be shared among the many elements of XHTML. For example, the `coreattrs` parameter entity contains a list of core attributes used by most XHTML elements. The `StyleSheet` and `Text` parameter entities contain code that sets the data types for style sheets and title attributes, respectively. From the DTD, you can quickly see that the four core attributes of XHTML are the `id`, `class`, `style`, and `title` attributes. You can also see that although they are available to almost all elements in the XHTML language, they are not required attributes. The specifications for the **br** element are fairly simple. Other elements of the XHTML language have more complicated rules, but all are based on the principles discussed in this tutorial.

**INSIGHT****Advantages and Disadvantages of DTDs**

DTDs are the common standard for validating XML documents, but they do have some serious limitations. Because a DTD is not written in the XML language, XML parsers must support the syntax and language requirements needed to interpret DTD code. DTDs are also limited in the data types they support. For example, you cannot specify that the value of an element or an attribute be limited only to integers or to text strings entered in a specific format; nor can you specify the structure of a document beyond a general description of the number or choice of child elements. Finally, DTDs do not support namespaces and thus are of limited value in compound documents.

However, DTDs are a recognized and long-supported standard. Therefore, you should have little problem finding parsers and applications to validate your XML documents based on DTDs. DTDs also support entities, providing a mechanism for referencing nontextual content from your XML files.

If the limitations of DTDs are a severe hindrance to validating your XML documents, another standard—schemas—provides support for an extended list of data types and can be easily adapted to compound documents involving several namespaces.

You've completed your work on Benjamin's document. He will use the DTD you developed to ensure that any new data added to the orders.xml document conforms to the standards you established.

**REVIEW****Session 2.3 Quick Check**

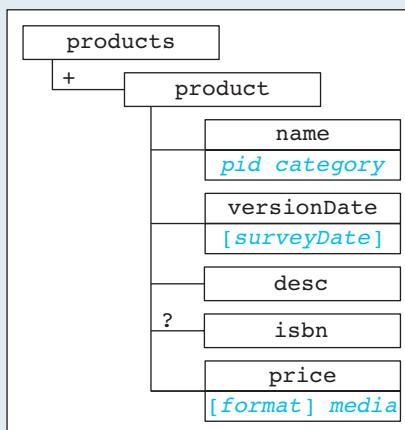
1. What is the difference between a general entity and a parameter entity?
2. What is the difference between a parsed entity and an unparsed entity?
3. What declaration stores the text string <Title>Hamlet</Title> as a general entity named `Play`? What command references this entity in a document?
4. What declaration stores the contents of the plays.xml file as a general entity named `Plays`?
5. What code stores the contents of the plays.dtd file as a parameter entity named `Works`?
6. What is a notation?
7. How do you reference the image file shakespeare.gif in an unparsed entity named `Portrait`? Assume that this entity is using a notation named `GIF`.

## Review Assignments

### Data File needed for the Review Assignments: prodtxt.xml

Benjamin needs your help with a document that lists some of Map Finds For You's map products. Figure 2-32 shows the tree structure of Benjamin's XML document.

**Figure 2-32** Structure of the products document



The document contains a root element named `products` with one or more occurrences of the `product` element containing information on map products. The `product` element contains five child elements—`name`, `versionDate`, `desc`, an optional `isbn`, and `price`. The `name` element stores the name of the product and supports two attributes—`pid`, the ID number of the product; and `category`, the type of product (`historical`, `state`, or `parks`). The `versionDate` element also supports an optional `surveyDate` attribute that indicates the date of the map survey, if known. The `desc` element stores a description of the product. The optional `isbn` element stores the ISBN for the product. The `price` element stores the name of the product and supports two attributes—`format`, the format of the product (`flat` or `raised`) with a default of `flat`; and `media`, the media type of the product (`paper` or `electronic`).

For this document, Benjamin wants to enforce a document structure to ensure that information recorded in the document is valid. Therefore, your task will be to create the DTD for the document.

Complete the following:

- Using your text editor, open the `prodtxt.xml` file from the `xml02 ▶ review` folder provided with your Data Files, enter `your name` and `today's date` in the comment section of the file, and then save the file as `products.xml`.
- In the `products.xml` file, insert an internal DTD for the root element `products` directly after the comment section and before the opening `<products>` tag.
- Within the internal DTD, declare the following items:
  - The `products` element, containing at least one occurrence of the child element `product`
  - The `product` element, containing five child elements in the sequence `name`, `versionDate`, `desc`, an optional `isbn`, and `price`
  - The `name`, `versionDate`, `desc`, `isbn`, and `price` elements, each containing parsed character data
- Add the following attribute declarations to the `product` file:
  - For the `name` element, a required `pid` attribute as an ID
  - For the `name` element, a required `category` attribute equal to `historical`, `state`, or `parks`

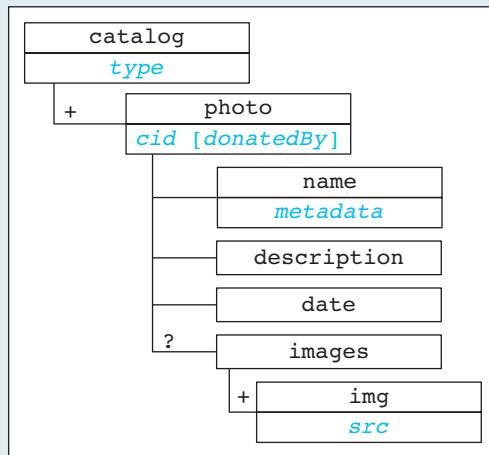
- c. For the `versionDate` element, an optional `surveyDate` attribute containing the date of the survey
  - d. For the `price` element, an optional `format` attribute equal to either `flat` or `raised`, with a default of `flat`
  - e. For the `price` element, a required `media` attribute equal to either `paper` or `electronic`
5. On the same line as the `format` attribute definition, insert a comment containing the text **format default is flat**.
6. Save your changes to the `products.xml` file, and then use Exchanger XML Editor or another XML tool to verify that the document is well formed.
7. Validate the document. If necessary, correct errors one at a time and revalidate until the document is valid.

**APPLY****Case Problem 1****Data Files needed for this Case Problem:** photostxt.dtd, photostxt.xml

Note: To complete this case problem, you need an XML parser capable of validating an XML document based on an external DTD file, such as Exchanger XML Editor.

**The Our Lady of Bergen Historical Society** Sharon Strattan is an archivist at the Our Lady of Bergen Historical Society in Bergenfield, New Jersey. The historical society is exploring how to transfer its listings to XML format, and Sharon has begun by creating a sample document of the society's extensive collection of photos. A schematic of the vocabulary she's developing is shown in Figure 2-33.

**Figure 2-33 Structure of the photos document**



The vocabulary Sharon designed has a root element named `catalog` containing one or more `photo` elements. Each `photo` element contains the name of the photo, a description, the estimated date the photo was taken, and, in some cases, a list of image files containing scans of the original photo. Sharon also added attributes to indicate the type of collection the photos come from, the collection ID for each photo, who donated the photo, a list of keywords (metadata) associated with the photo, and the source of any image file. You'll assist Sharon by creating a DTD based on her XML vocabulary, and then you'll use the DTD to validate her sample document.

Complete the following:

1. Using your text editor, open `photostxt.dtd` and `photostxt.xml` from the `xml02 ▶ case1` folder provided with your Data Files, enter **your name** and **today's date** in the comment section of each file, and then save the files as `photos.dtd` and `photos.xml`, respectively.

2. In the photos.dtd file, declare the following elements:
  - a. The **catalog** element, containing one or more **photo** elements
  - b. The **photo** element, containing the following sequence of child elements—**name**, **description**, **date**, and (optionally) **images**
  - c. The **name**, **description**, and **date** elements, containing only parsed character data
  - d. The **images** element, containing one or more **img** elements
  - e. The **img** element, containing empty content
3. Declare the following attributes in the DTD:
  - a. The **type** attribute, a required attribute of the **catalog** element, containing a valid XML name (*Hint:* Use the NMTOKEN data type.)
  - b. The **cid** attribute, a required ID attribute of the **photo** element
  - c. The **donatedBy** attribute, an optional attribute of the **photo** element, containing character data
  - d. The **metadata** attribute, a required attribute of the **name** element, containing a list of valid XML names (*Hint:* Use the NMTOKENS data type.)
  - e. The **src** attribute, a required attribute of the **img** element, containing character data
4. Save your changes to the photos.dtd file.
5. In the photos.xml file, directly after the comment section, insert a DOCTYPE that references the system location photos.dtd.
6. Save your changes to the photos.xml file.
7. Verify that the photos.xml file is well formed, and then validate it. Revalidate after correcting each error in the code, if necessary, until the document passes validation. (*Hint:* Because this document uses an external DTD, you must correct any validation errors related to the DTD in the photos.dtd file.) Note that you cannot use <http://validator.w3.org> to validate an XML file against a nonpublic external DTD file, so you must use a program such as Exchanger to validate this file.

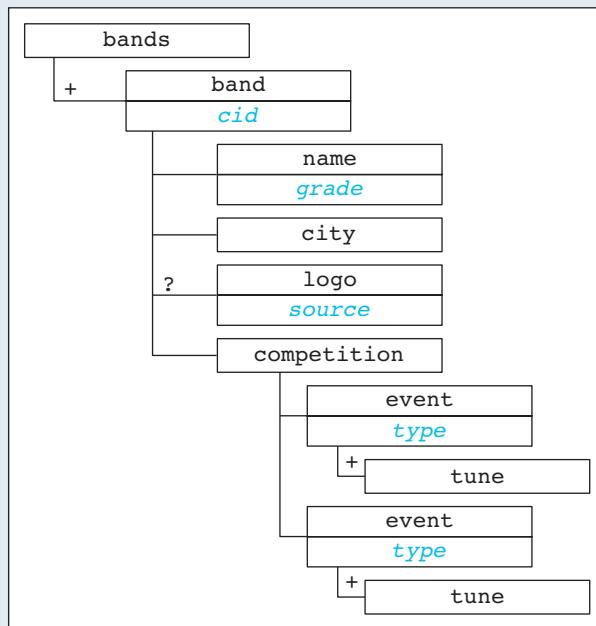
APPLY

## Case Problem 2

Data File needed for this Case Problem: **mdpbatxt.xml**

**Midwest Developmental Pipe Band Association** Jacob St. John works as a coordinator for the Midwest Developmental Pipe Band Association (MDPBA), and is responsible for coordinating competitions for the MDPBA's many developmental pipe bands in the midwestern United States. A pipe band is a musical group consisting of pipe players and drummers. Pipe bands are traditional in Great Britain, as well as in other parts of the world that have received British cultural influence. Part of Jacob's job is to maintain a document that lists competition entries for each pipe band. Jacob has asked for your help in creating XML documents that maintain a consistent document structure. He has created a sample document describing five pipe bands. The document lists the competition program for each pipe band. Each program includes exactly two events in which each band plays one or more tunes. One event type is a March, Strathspey, & Reel (MSR). The other event type is a Medley event. Figure 2-34 shows a tree diagram for the vocabulary.

**Figure 2-34 Structure of the mdpba document**



The document also contains an optional `logo` element that stores information about graphic files of the pipe band logos. Any DTD you create for this document also must work with the unparsed data contained in these graphic files.

Complete the following:

1. Using your text editor, open `mdpbatxt.xml` from the `xml02 ▶ case2` folder provided with your Data Files, enter **your name** and **today's date** in the comment section of the file, and then save the file as `mdpba.xml`.
2. Review the contents of the `mdpba.xml` file. Directly after the comment section, insert a DOCTYPE that includes DTD statements for the following elements:
  - a. The `bands` element, containing at least one occurrence of the child element `band`
  - b. The `band` element, containing child elements in the sequence `name`, `city`, `logo` (optional), and `competition`
  - c. The `name` and `city` elements, containing parsed character data
  - d. The `logo` element as an empty element
  - e. The `competition` element, containing exactly two child elements named `event`
  - f. The `event` element, containing at least one occurrence of the child element `tune`
  - g. The `tune` element, containing parsed character data
3. Declare the following required attributes in the DTD:
  - a. The `band` element should contain a single required attribute named `cid` containing an ID value.
  - b. The `name` element should contain a required attribute named `grade` with values limited to `1, 2, 3, 4, 5, juvenile, and novice`.
  - c. The `logo` element should contain a required entity attribute named `source`.
  - d. The `event` element should contain a required attribute named `type` with values limited to `MSR or Medley`.
4. Declare a notation named `JPG` with a system location equal to `image/jpg`.
5. Create two unparsed entities within the internal subset to the DTD. Each entity should reference the `JPG` notation. The first entity should be named `celtic` and reference the `celtic.jpg` file. The second entity should be named `badger` and reference the `badger.jpg` file.
6. Save your changes to the `mdpba.xml` file.
7. Verify that the document is well formed and then validate it.

## Case Problem 3

### Data Files needed for this Case Problem: **donors.txt**, **rostertxt.xml**, **seatrusttxt.dtd**

Note: To complete this case problem, you need an XML parser capable of validating an XML document based on an external DTD file, such as Exchanger XML Editor.

**The Save Exotic Animals Trust** Sienna Woo is the donor coordinator for the Save Exotic Animals Trust (SEA Trust), a charitable organization located in central Florida. One of her responsibilities is to maintain a membership list of people in the community who have donated to SEA Trust. Donors can belong to one of four categories—Friendship, Patron, Sponsor, or Founder. The categories assist Sienna in marketing SEA Trust’s fundraising goals and in developing strategies to reach those goals.

Currently, most of the data that Sienna has compiled resides in text files. To make the fundraising campaign strategies more effective, she wants to convert this data into an XML document and ensure that the resulting document follows some specific guidelines. You will create the XML document for her.

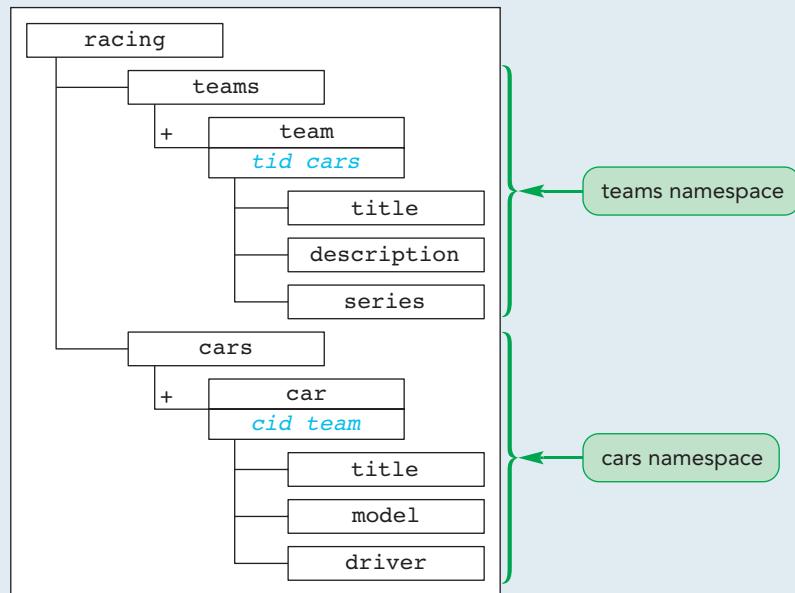
Complete the following:

1. Using your text editor, open the **rostertxt.xml** and **seatrusttxt.dtd** files from the **xml02 ▶ case3** folder provided with your Data Files, insert *your name* and *today's date* in the comment section of each file, and then save the files as **roster.xml** and **seatrust.dtd**, respectively.
2. Add the data stored in the **donors.txt** file in the **xml02 ▶ case3** folder to the **roster.xml** file as the document content, and then add XML elements to structure the data in the **roster.xml** file as follows: (Note: You should ignore any validation or well-formedness errors flagged by your editor until the document is finished.)
  - a. A root element named **roster** should contain several **donor** elements.
  - b. Each **donor** element should contain the following child elements, which should appear no more than once within the **donor** element, except as noted—**name**, **address**, **phone** (one or more), **email** (optional), **donation**, **method**, and **effectiveDate**.
  - c. The **phone** element should contain an attribute named **type** that identifies the phone type—**home**, **work**, or **cell**. This should be a required attribute for each **phone** element.
  - d. The **donor** element should contain an attribute named **level** that identifies the donor level—**friendship**, **patron**, **sponsor**, or **founder**. This should be a required attribute for each **donor** element.
3. In the **seatrust.dtd** file, create a DTD based on the structure you created in the **roster.xml** file. Save and close the **seatrust.dtd** file.
4. Apply your DTD to the contents of the **roster.xml** file. Save your changes to the **roster.xml** file.
5. Verify that the **roster.xml** file is well formed and valid.

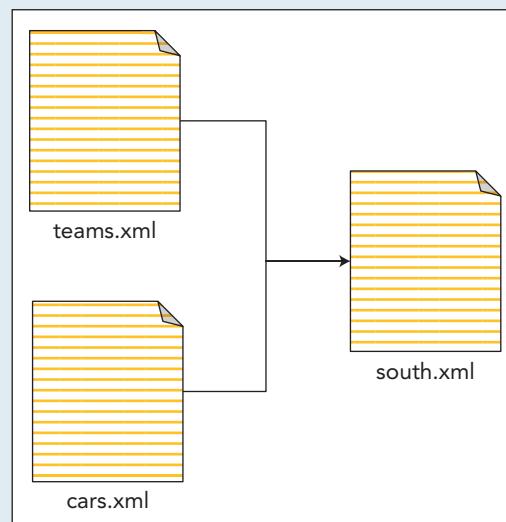
## Case Problem 4

### Data Files needed for this Case Problem: **carstxt.xml**, **southtxt.xml**, **teamstxt.xml**

**South Racing** Danika Francis tracks team cars for South Racing’s racing teams. As part of her job, she has created several XML vocabularies dealing with team series and the cars available to race in them. She wants to create a compound document combining information from the teams and cars vocabularies. She also wants to validate any data entered into her documents, and she asks you to develop a DTD. Because she is creating compound documents that combine elements from the teams and cars namespaces, she needs the DTD to work with namespaces. Figure 2-35 shows the tree structure of the compound document.

**Figure 2-35 Structure of the compound document**

Danika wants you to create a master XML document containing a DTD for the teams and cars namespaces, and then use entities to read the information from different XML documents into the master compound document. Figure 2-36 shows a schematic of the file relationships in Danika's proposed project.

**Figure 2-36 File relationships for the compound document**

Complete the following:

1. Using your text editor, open the **carstxt.xml**, **southtxt.xml**, and **teamstxt.xml** files from the **xml02 ▶ case4** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **cars.xml**, **south.xml**, and **teams.xml**, respectively.
2. In the **teams.xml** file, place all the elements in the namespace **http://example.com/southracing/ teams** with the namespace prefix **t**. Save your changes to the file.
3. In the **cars.xml** file, place all the elements in the namespace **http://example.com/southracing/ cars** with the namespace prefix **c**. Save your changes to the file.

 **EXPLORE** 4. In the south.xml file, insert an internal DTD subset with the following element declarations (making sure to include the namespace prefix `t` with all of the element names):

- The `teams` element, containing at least one child element named `team`
- The `team` element, containing the following sequence of child elements: `title`, `description`, and `series`
- The `title`, `description`, and `series` elements, containing parsed character data

 **EXPLORE** 5. Add an attribute declaration to the `teams` element to declare the `http://example.com/southracing/teams` namespace as a fixed value.

- Add the following attribute declarations to the `team` element:
  - An ID attribute named `tid`
  - An attribute named `cars`, containing a list of ID references

 **EXPLORE** 7. Add the following element declarations (making sure all element references include the `c` namespace prefix):

- The `cars` element, containing at least one child element named `car`
- The `car` element, containing the following sequence of child elements: `title`, `model`, and `driver`
- The `title`, `model`, and `driver` elements, containing parsed character data

 **EXPLORE** 8. Add an attribute declaration to the `cars` element to declare the `http://example.com/southracing/cars` namespace as a fixed value.

- Add the following attribute declarations to the `car` element:
  - An ID attribute named `cid`
  - A `team` attribute containing a list of ID references
- Add the root element `racing` to the document belonging to the default namespace `http://example.com/southracing`.

 **EXPLORE** 11. In the internal DTD subset, add the following declarations:

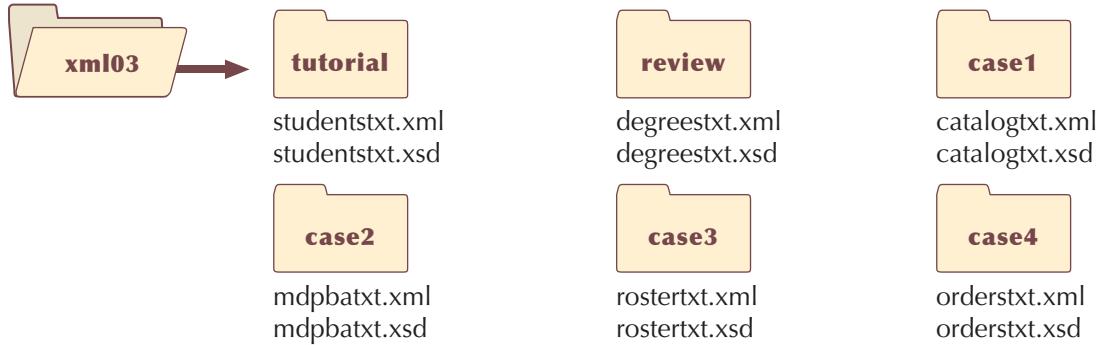
- The `racing` element, containing the two child elements `t:teams` and `c:cars`
  - A fixed attribute of the `racing` element declaring the `http://example.com/southracing` namespace
  - An external entity named `teamsList` pointing to the `teams.xml` file
  - An external entity named `carsList` pointing to the `cars.xml` file
- Within the `racing` element, insert references to the `teamsList` and `carsList` entities.
  - Save your changes to the `south.xml` file, and then close it.
  - Verify that all the documents are well formed and valid.

**OBJECTIVES****Session 3.1**

- Compare schemas and DTDs
- Explore different schema vocabularies
- Declare simple type elements and attributes
- Declare complex type elements
- Apply a schema to an instance document

**Session 3.2**

- Work with XML Schema data types
- Derive new data types for text strings, numeric values, and dates
- Create data types for patterned data using regular expressions

**STARTING DATA FILES**

# Validating Documents with Schemas

*Creating a Schema for the ATC School of Information Technology*

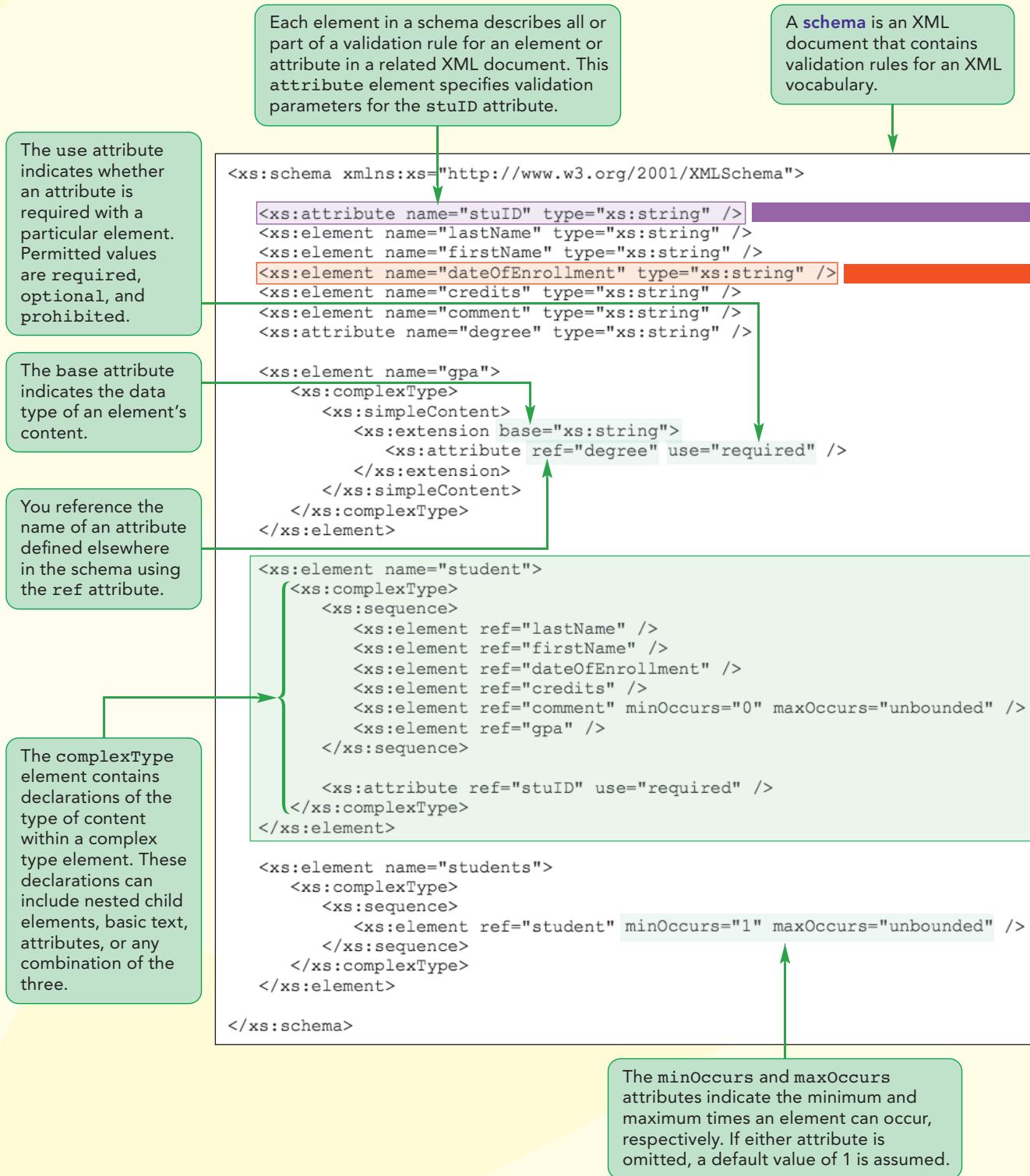
## Case | ATC School of Information Technology

Sabrina Lincoln is an academic advisor for the School of Information Technology at Austin Technical College (ATC) in Austin, Utah, where she advises students in the information technology programs. Sabrina wants to use XML to create structured documents containing information on the different programs and the students enrolled in those programs. Eventually, the XML documents can be used as a data resource for the center's intranet, enabling faculty advisors to view program and student data online.

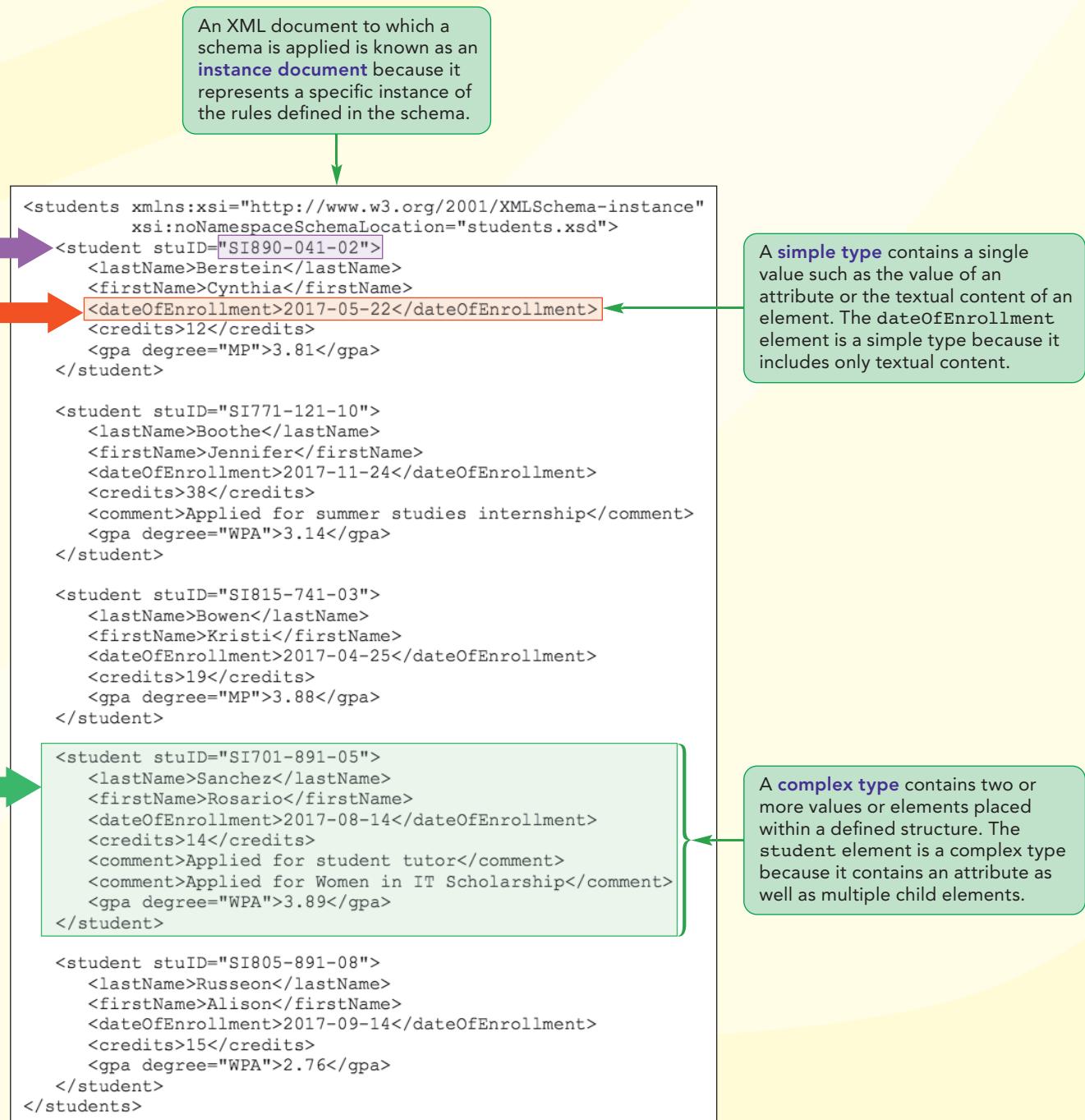
Accuracy is important to ATC, and Sabrina needs to know that the data she enters is error free. In particular, she must be able to confirm that the student data in her XML documents matches the criteria for the programs. Sabrina also needs to create documents from the various XML vocabularies she's created. For example, she may need to create a document that combines student information with information on the programs themselves.

DTDs cannot fulfill Sabrina's needs because DTDs have a limited range of data types and provide no way to deal with numeric data. Also, DTDs and namespaces do not mix well. However, schemas can work with a wide range of data types and do a better job of supporting namespaces than DTDs. In this tutorial, you'll develop schemas for the XML document that Sabrina has created.

# Session 3.1 Visual Overview:



# Structure of a Basic Schema



## Introducing XML Schema

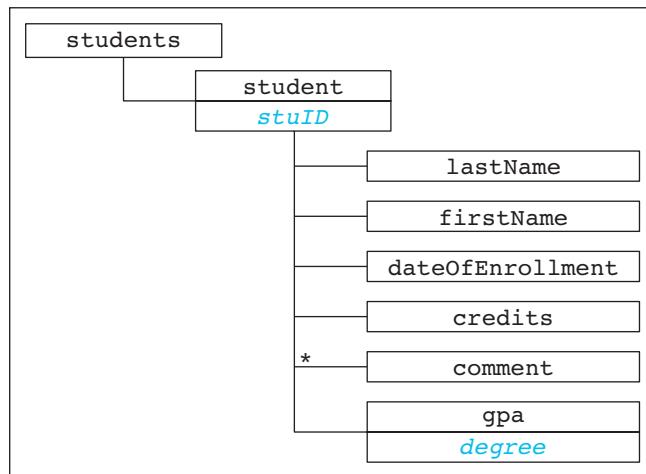
You and Sabrina meet at the School of Information Technology to discuss her work for Austin Technical College. She has brought along a file named `students.xml`, which contains a list of students accepted into programs within the school. You'll open this file now.

### To open the `students.xml` document:

- 1. Use your text editor to open `studentstxt.xml` from the `xml03 ▶ tutorial` folder provided with your Data Files, enter **your name** and **today's date** in the comment section, and then save the file as `students.xml`.
- 2. Examine the contents of the document, paying close attention to the order of the elements and the values of the elements and attributes.

Figure 3-1 shows the tree structure of the XML vocabulary used in the document. For each `student` element in the document, Sabrina inserted the attribute `stuID`. The attribute contains the student's assigned student number. In addition, Sabrina has collected each student's last and first names, date of enrollment, credits, and GPA. Each `student` element can also contain multiple `comment` elements for any additional information that an advisor wants to add.

Figure 3-1 Structure of the students vocabulary



The `students.xml` file contains information on a few students in the information technology programs, but eventually it will contain more entries. As more students are added, Sabrina wants the document to require that the data for each student meet eligibility guidelines for the specified program of study. For example, students' GPAs must be at least 2.0 on a 4.0 scale, and every student must have a valid student ID number. For the initial faculty advising rollout, students must be enrolled in either the Mobile Programmer or Web Programmer/Analyst programs. Your task is to set up a validation system for this document.

## The Limits of DTDs

DTDs are commonly used for validation largely because of XML's origins as an offshoot of SGML. SGML originally was designed for text-based documents, such as reports and technical manuals. As long as data content is limited to simple text, DTDs work well for validation. However, as XML began to be used for a wider range of document content, developers needed an alternative to DTDs.

One complaint about DTDs is their lack of data types. For example, Sabrina can declare a `gpa` element in the DTD, but she cannot specify that the `gpa` element may contain only numbers or that those numbers must fall within a specified range of values. Likewise, she can declare a `dateOfEnrollment` element, but a DTD cannot require that element to contain only dates. DTDs simply do not provide the control over data that Sabrina requires. DTDs also do not recognize namespaces, so they are not well suited to compound documents in which content from several vocabularies needs to be validated. This is a concern for Sabrina because her job at the ATC will involve developing several XML vocabularies that often will be combined into a single document.

Finally, DTDs employ a syntax called **Extended Backus–Naur Form (EBNF)**, which is different from the syntax used for XML. This means that a document's author must be able to work not only with the syntax of XML, but with EBNF as well. For developers who want to work with only one language, this could be a concern.

Because of XML's extensibility, you can instead use XML itself to document the structure and content of other XML documents. This is the idea behind schemas.

## Schemas and DTDs

### TIP

You should validate a document using either a DTD or a schema but not both because some XML parsers validate a document under one and ignore the other, causing the document to be rejected.

A schema is an XML document that contains validation rules for an XML vocabulary. When applied to a specific XML file, the document to be validated is called the instance document because it represents a specific instance of the rules defined in the schema. Schemas have several advantages over DTDs. XML parsers need to understand only XML, so all the tools used to create an instance document can also be applied to designing the schema. Schemas also support more data types, including data types for numbers and dates as well as custom data types for special needs. Additionally, schemas are more flexible than DTDs in dealing with elements that contain both child elements and text content, and they provide support for namespaces, making it easier to validate compound documents. Figure 3-2 summarizes some of the most significant differences between schemas and DTDs.

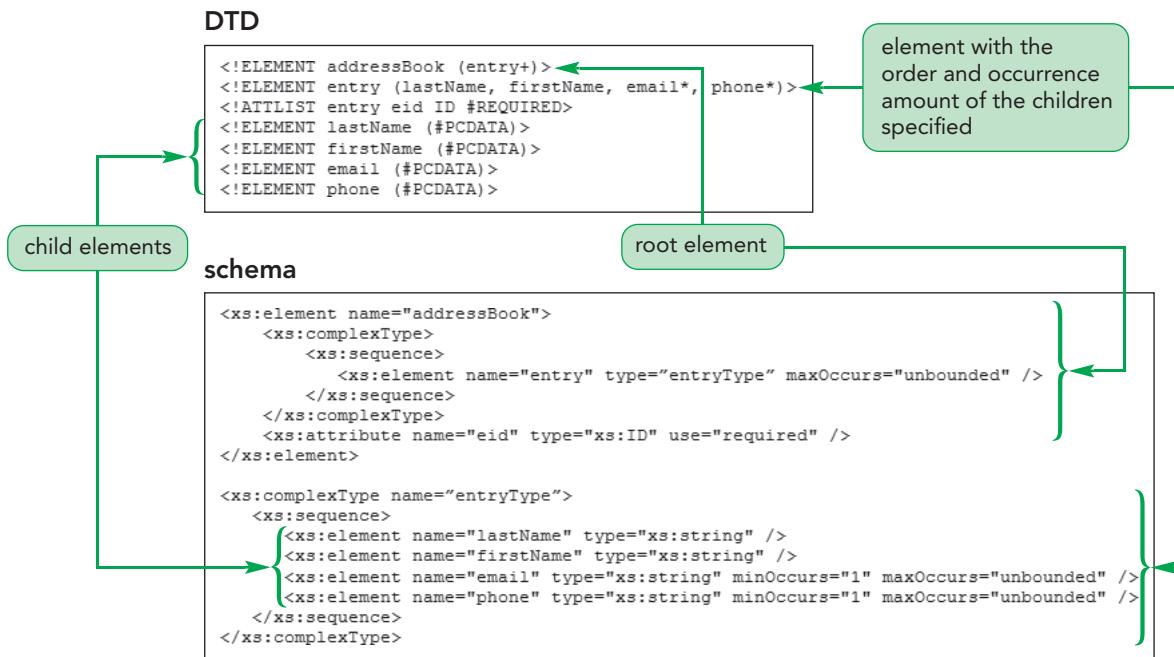
**Figure 3-2**

**Comparison of schemas and DTDs**

Feature	Schemas	DTDs
Document language	XML	Extended Backus Naur Form (EBNF)
Standards	multiple standards	one standard
Supported data types	44 (19 primitive + 25 derived)	10
Customized data types	yes	no
Mixed content	easy to develop	difficult to develop
Namespaces	completely supported	only namespace prefixes are supported
Entities	no	yes

If schemas are so useful, why do you need DTDs? First, DTDs represent an older standard for XML documents and are more widely supported. DTDs are simpler to create and maintain than schemas because the language itself is easier to work with. This is partly due to the fact that DTDs are more limited than schemas. Thus, DTDs are easier to set up for basic documents that don't require much validation. Figure 3-3 compares a simple DTD to its equivalent basic schema.

**Figure 3-3 Comparison of simple DTD and simple schema**



## Schema Vocabularies

Unlike DTDs, a single standard doesn't exist for schemas. Instead, several schema vocabularies have been created to serve the needs of different XML developers. Because schemas are written in XML, a **schema vocabulary** is simply an XML vocabulary created for the purpose of describing schema content. Figure 3-4 describes some schema vocabularies.

**Figure 3-4****Schema vocabularies**

<b>Schema</b>	<b>Description</b>
XML Schema	The most widely used schema standard, XML Schema is developed and maintained by the W3C, and is designed to handle a broad range of document structures. It is also referred to as XSD.
Document Definition Markup Language (DDML)	One of the original schema languages, DDML (originally known as XSchema) was created to replicate all DTD functionality in a schema. DDML does not support any data types beyond what could be found in DTDs.
XML Data	One of the original schema languages, XML Data was developed by Microsoft to replace DTDs.
XML Data Reduced (XDR)	XDR is a subset of the XML Data schema, and was primarily used prior to the release of XML Schema.
Regular Language description for XML (RELAX)	A simple alternative to the W3C's XML Schema standard, RELAX provides much of the same functionality as DTDs, with additional support for namespaces and data types. RELAX does not support entities or notations.
Tree Regular Expressions for XML (TREX)	A TREX schema specifies a pattern for an XML document's structure and content, and thus identifies a class of XML documents that match the pattern. TREX has been merged with RELAX into RELAX NG.
RELAX NG (Regular Language for XML Next Generation)	RELAX NG is the current version of RELAX, combining the features of RELAX and TREX.
Schematron	The Schematron schema represents documents using a tree pattern, allowing support for document structures that might be difficult to represent in traditional schema languages.

Support for a particular schema depends solely on the XML parser being used for validation. Before applying any of the schemas listed in Figure 3-4, you must verify the level of support offered by your application for that particular schema. XML Schema, developed by the W3C in March 2001, is the most widely adopted schema standard. Although this tutorial focuses primarily on XML Schema, many of the concepts involved with XML Schema can be applied to the other schema vocabularies.

## Starting a Schema File

A DTD can be placed within an instance document or within an external file. A schema, however, is always placed in an external file. XML Schema filenames end with the *.xsd* file extension. Sabrina has created a blank XML Schema file for you to work on. You'll open her file now.

### To start work on the XML Schema file:

- 1. Use your text editor to open **studentstxt.xsd** from the **xml03 ▶ tutorial** folder, and then enter **your name** and **today's date** in the comment section.  
**Trouble?** If you open the file in an XML editor, such as Exchanger, you may see an error message indicating that the document is not well formed and contains a premature end of file. In later steps, after you add your first validation rule to this file, the error should be resolved. For now, there's no need to worry about it.
- 2. Save the file as **students.xsd**.

The root element in any XML Schema document is the `schema` element. For a parser to recognize that a document is written in the XML Schema vocabulary, the `schema` element must include a declaration for the XML Schema namespace using the URI <http://www.w3.org/2001/XMLSchema>. The general structure of an XML Schema file is

```
<?xml version="1.0" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    content
</schema>
```

where `content` is the list of elements and attributes that define the rules of the instance document. By convention, the namespace prefix `xsd` or `xs` is assigned to the XML Schema namespace to identify elements and attributes that belong to the XML Schema vocabulary. Keeping well-defined namespaces in an XML Schema document becomes very important when you start creating schemas for compound documents involving several namespaces. Therefore, the usual form of an XML Schema document is

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    content
</xs:schema>
```

This tutorial assumes a namespace prefix of `xs` when discussing the elements and attributes of the XML Schema language. However, you can choose to use a different prefix in your own XML Schema documents. You can also set XML Schema as the document's default namespace, which eliminates the need for a prefix. The only requirement is that you be consistent in the use of a namespace prefix.

## REFERENCE

### *Creating a Schema*

- To create an XML Schema document, insert the structure

```
<?xml version="1.0" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    content
</schema>
```

in the file, where `content` consists of the XML Schema elements and attributes used in defining the rules for the instance document.

- To apply a namespace prefix (customarily `xs` or `xsd`) to the elements and attributes of the XML Schema vocabulary, use the following structure:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    content
</xs:schema>
```

You'll add the root `schema` element to the `students.xsd` file now, using the `xs` namespace prefix to place it in the XML Schema namespace.

### **To insert the schema element in the `students.xsd` file:**

- 1. Directly below the comment section, insert the following code:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xs:schema>
```

Figure 3-5 shows the `schema` element in the `students.xsd` file.

Figure 3-5

### XML Schema root element

XML Schema  
namespace URI

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
    New Perspectives on XML
    Tutorial 3
    Tutorial Case

    Austin Technical College information technology schema
    Author: Sabrina Lincoln
    Date: 12/9/2017|
```

Filename: students.xsd  
Supporting Files: students.xml  
-->

```
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xsschema>
```

- 2. Save your changes to the file.

## Understanding Simple and Complex Types

XML Schema supports two types of content—simple and complex. A simple type contains only text and no nested elements. Examples of simple types include all attributes, as well as elements with only textual content. A complex type contains two or more values or elements placed within a defined structure. Examples of complex types include an empty element that contains an attribute, and an element that contains child elements. Figure 3-6 shows examples of both simple and complex types.

Figure 3-6

### Simple and complex types

Simple Types	Complex Types
<b>An element containing only text</b> <student>Cynthia Berstein</student>	<b>An empty element containing attributes</b> <student name="Cynthia Berstein" stuID="SI890-041-02" />
<b>An attribute</b> stuID="SI890-041-02"	<b>An element containing text and an attribute</b> <student stuID="SI890-041-02">Cynthia Berstein</student>
	<b>An element containing child elements</b> <student><name>Cynthia Berstein</name><stuID>SI890-041-02</stuID></student>
	<b>An element containing child elements and an attribute</b> <student stuID="SI890-041-02"><firstName>Cynthia</firstName><lastName>Berstein</lastName></student>

The `students.xml` file contains several examples of simple and complex types, which are listed in Figure 3-7. Note that all attributes in the document are, by default, simple types. The `students`, `student`, and `gpa` elements are complex types because they contain either nested child elements or attributes. The `lastName`, `firstName`, `dateOfEnrollment`, `credits`, and `comment` elements are all simple types because each contains only element text.

**Figure 3-7** Simple and complex types in the students.xml document

Item	Contains	Content Type
students	nested child elements	complex
student	nested child elements	complex
stuID	an attribute value	simple
lastName	element text	simple
firstName	element text	simple
dateOfEnrollment	element text	simple
credits	element text	simple
comment	element text	simple
gpa	an attribute	complex
degree	an attribute value	simple

The distinction between simple and complex types is important in XML Schema because the code to define a simple type differs greatly from the code to define a complex type. You'll start writing the schema for Sabrina's document by defining all of the simple types found in the students.xml file.

## Defining a Simple Type Element

An element in the instance document containing only text and no attributes or child elements is defined in XML Schema using the `<xs:element>` tag

```
<xs:element name="name" type="type" />
```

where `name` is the name of the element in the instance document and `type` is the type of data stored in the element. The data type can be one of XML Schema's built-in data types, or it can be a data type defined by the schema author. If you use a built-in data type, you must indicate that it belongs to the XML Schema namespace because it is a feature of the XML Schema language. Therefore, the code to use a built-in data type is

```
<xs:element name="name" type="xs:type" />
```

where `type` is a data type supported by XML Schema. Note that if you use a different namespace prefix or declare XML Schema as the default namespace for the document, the prefix will be different.

Perhaps the most commonly used data type in XML Schema is `string`, which allows an element to contain any text string. For example, in the students.xml file, the `lastName` element contains the text of the student's last name. To indicate that this element contains string data, you would add the following element to the XML Schema file:

```
<xs:element name="lastName" type="xs:string" />
```

Another popular data type in XML Schema is `decimal`, which allows an element to contain a decimal number.

## REFERENCE

**Defining a Simple Type Element**

- To define a simple type element, enter

```
<xs:element name="name" type="type" />
```

where *name* is the element name in the instance document and *type* is the data type.

- To use a data type built into the XML Schema language, place *type* in the XML Schema namespace as follows:

```
<xs:element name="name" type="xs:type" />
```

For now, you'll define the data type of each simple type element as a simple text string. You'll revise these declarations in the next session, when you examine the wide variety of data types supported by XML Schema as well as learn how to define your own data types.

**To declare the simple type elements:**

- Within the schema root element, insert the following simple type elements, making sure to match the case:

```
<xs:element name="lastName" type="xs:string" />
<xs:element name="firstName" type="xs:string" />
<xs:element name="dateOfEnrollment" type="xs:string" />
<xs:element name="credits" type="xs:string" />
<xs:element name="comment" type="xs:string" />
```

Figure 3-8 shows the inserted simple type elements.

Figure 3-8

**Elements defined as simple types**

- Save your changes to the file.

**Defining an Attribute**

The other simple type content found in Sabrina's document consists of attribute values. To define an attribute in XML Schema, you use the `<xs:attribute>` tag

```
<xs:attribute name="name" type="type" default="default" fixed="fixed" />
```

where *name* is the name of the attribute, *type* is the data type, *default* is the attribute's default value, and *fixed* is a fixed value for the attribute. The *default* and *fixed* attributes are optional. You use the *default* attribute to specify a default attribute value,

which is applied when no attribute value is entered in the instance document; you use the *fixed* attribute to fix an attribute to a specific value.

Attributes use the same collection of data types that simple type elements do. For example, the following code defines the *degree* attribute and indicates that it contains a text string with a default value of WPA:

```
<xs:attribute name="degree" type="xs:string" default="WPA" />
```

**REFERENCE**

### Defining an Attribute

- To define an attribute, use the syntax

```
<xs:attribute name="name" type="type" default="default"  
fixed="fixed" />
```

where *name* is the attribute name in the instance document, *type* is the data type of the attribute, *default* specifies a default value for the attribute when no attribute value is entered in the instance document, and *fixed* fixes the attribute to a specific value. The *default* and *fixed* values are optional.

- For data types that are part of the XML Schema vocabulary, place *type* in the XML Schema namespace, as follows:

```
<xs:attribute name="name" type="xs:type" default="default"  
fixed="fixed" />
```

The students.xml file uses two attributes—stuID and degree. Neither of these attributes has a default value or a fixed value. You'll add the attribute declarations to the schema file below the element declarations you just created.

### To define the attributes used in the students.xml file:

- 1. Add the following attribute definition on a new line above the first element definition:  

```
<xs:attribute name="stuID" type="xs:string" />
```

As in a DTD, the declarations in a schema can be placed in any order. However, you can make your code easier to understand by keeping your declarations organized. Because the second attribute you'll declare occurs near the end of the XML code for each record, you'll place it below the other declarations.
- 2. Add the following attribute definition on a new line below the last element definition:  

```
<xs:attribute name="degree" type="xs:string" />
```

Your completed code should match Figure 3-9.

Figure 3-9

### Attributes defined as simple types

attribute definitions  
can be grouped  
together or separated

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:attribute name="stuID" type="xs:string" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="dateOfEnrollment" type="xs:string" />
    <xs:element name="credits" type="xs:string" />
    <xs:element name="comment" type="xs:string" />
    <xs:attribute name="degree" type="xs:string" />
</xs:schema>
```

attribute contains only  
a simple text string

- 3. Save your changes to the file.

## Defining a Complex Type Element

The two attributes you defined are not yet associated with any elements in Sabrina's document. To create those associations, you must first define the element containing each attribute. Because those elements contain attributes, they are considered complex type elements. The basic structure for defining a complex type element with XML Schema is

```
<xs:element name="name">
    <xs:complexType>
        declarations
    </xs:complexType>
</xs:element>
```

where *name* is the name of the element and *declarations* represents declarations of the type of content within the element. This content could include nested child elements, basic text, attributes, or any combination of the three. As shown in Figure 3-6, the following four complex type elements usually appear in an instance document:

- An empty element containing only attributes
- An element containing text content and attributes but no child elements
- An element containing child elements but no attributes
- An element containing both child elements and attributes

XML Schema uses a different code structure for each of these four possibilities. You'll start by looking at the definition for an empty element that contains one or more attributes.

## Defining an Element Containing Only Attributes

The code to define the attributes of an empty element is

```
<xs:element name="name">
    <xs:complexType>
        attributes
    </xs:complexType>
</xs:element>
```

where *name* is the name of the empty element in the instance document and *attributes* is the set of simple type elements that define the attributes associated with the empty element. For example, the empty **student** element

```
<student name="Cynthia Bernstein" gpa="3.81"/>
```

has two attributes—*name* and *gpa*. The code for this complex type element has the following structure:

```
<xs:element name="student">
    <xs:complexType>
        <xs:attribute name="name" type="xs:string" />
        <xs:attribute name="gpa" type="xs:decimal" />
    </xs:complexType>
</xs:element>
```

The order of the attribute declarations is unimportant. XML Schema allows attributes to be entered in any order within a complex type element.

## Defining an Element Containing Attributes and Basic Text

If an element in the instance document contains attributes and text content but no child elements, the structure that declares the element and the attributes takes a different form. In these cases, the definition needs to indicate that the element contains simple content and a collection of one or more attributes. The structure of the element definition is

```
<xs:element name="name">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="type">
                attributes
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
```

where *type* is the data type of the element's content (or *xs:type* if the data type is part of the XML Schema vocabulary) and *attributes* represents a list of the attributes associated with the element. The purpose of the **simpleContent** element in this code is to indicate that the element contains only text and no nested child elements. The **<xs:extension>** tag is used to extend this definition to include the list of attributes. The **simpleContent** and **extension** elements are important tools used by XML Schema to derive new data types and to define complex content. For the *students.xml* document, you'll use them to define the **gpa** element, which is a complex type element that contains a text value and is associated with an attribute. The following is a sample of the type of content stored in this element:

```
<gpa degree="WPA">3.81</gpa>
```

The following code defines this element and associates it with the `degree` attribute:

```
<xs:element name="gpa">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="degree" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

In this code, the `base` attribute in the `<xs:extension>` element sets the data type for the `gpa` element. At this point, you're assuming that the `gpa` element contains a text string. This code also sets the data type of the `degree` attribute to `xs:string`, indicating that it contains a text string.

## Referencing an Element or Attribute Definition

You've already defined the `degree` attribute in the `students.xsd` file. You could revise the code to nest that attribute definition within the definition of the `gpa` element. However, XML Schema allows for a great deal of flexibility in writing complex types.

Rather than repeating that earlier attribute declaration within the `gpa` element, you can create a reference to it. The code to create a reference to an element definition is

```
<xs:element ref="elemName" />
```

where `elemName` is the name used in the element definition. Likewise, the code to create a reference to an attribute definition is

```
<xs:attribute ref="attName" />
```

where `attName` is the name used in the attribute definition. The following code defines the `degree` attribute and then references the definition from within the `gpa` element:

```
<xs:attribute name="degree" type="xs:string" />

<xs:element name="gpa">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="degree" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

### TIP

To repeat an attribute or element definition, simplify the schema by defining the attribute or element once, and then reference that definition wherever it is used.

### Defining a Complex Type Element

- To define an empty element containing one or more attributes, use

```
<xs:element name="name">
    <xs:complexType>
        attributes
    </xs:complexType>
</xs:element>
```

where *name* is the element name and *attributes* represents a list of attributes associated with the element.

- To define an element containing text content and one or more attributes, use

```
<xs:element name="name">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="type">
                attributes
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
```

where *type* is the data type of the text content of the element.

- To define an element containing only nested child elements, use

```
<xs:element name="name">
    <xs:complexType>
        <xs:compositor>
            elements
        </xs:compositor>
    </xs:complexType>
</xs:element>
```

where *elements* is a list of the child elements, and *compositor* is *sequence*, *choice*, or *all*.

- To define an element containing both attributes and nested child elements, use

```
<xs:element name="name">
    <xs:complexType>
        <xs:compositor>
            elements
        </xs:compositor>
        attributes
    </xs:complexType>
</xs:element>
```

You'll add the definition of the *gpa* element to the *students.xsd* file now.

### To define the gpa element:

- 1. In the students.xsd file, below the declaration for the degree attribute, add the following complex type declaration:

```
<xs:element name="gpa">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="degree" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Figure 3-10 shows the complex type declaration inserted in the schema element.

Figure 3-10

### Complex type element containing text and an attribute

ref attribute value  
uses name defined  
for the degree  
attribute

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="stuID" type="xs:string" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="dateOfEnrollment" type="xs:string" />
  <xs:element name="credits" type="xs:string" />
  <xs:element name="comment" type="xs:string" />
  <xs:attribute name="degree" type="xs:string" />

  <xs:element name="gpa">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute ref="degree" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

the text content of  
the gpa element is  
a simple text string

- 2. Save your changes to the file.

## Defining an Element with Nested Children

Next, you'll examine complex elements that contain nested child elements but no attributes or text. To define this type of complex element, you use the structure

```
<xs:element name="name">
  <xs:complexType>
    <xs:compositor>
      elements
    </xs:compositor>
  </xs:complexType>
</xs:element>
```

where `name` is the name of the element, `compositor` is a value that defines how the child elements appear in the document, and `elements` is a list of the nested child elements. You can choose any of the following compositors to define how the child elements display in the document:

- **sequence**—requires the child elements to appear in the order listed in the schema
- **choice**—allows any *one* of the child elements listed to appear in the instance document
- **all**—allows any of the child elements to appear in any order in the instance document; however, each may appear only once, or not at all

For example, the following code assigns four child elements—`street`, `city`, `state`, and `country`—to the `address` element:

```
<xs:element name="address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="street" type="xs:string" />
      <xs:element name="city" type="xs:string" />
      <xs:element name="state" type="xs:string" />
      <xs:element name="country" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Because the definition uses the `sequence` compositor, the document is invalid if the `address` element doesn't contain all the listed child elements in the order specified.

The following definition allows the `sponsor` element to contain an element named `parent` or an element named `guardian`:

```
<xs:element name="sponsor">
  <xs:complexType>
    <xs:choice>
      <xs:element name="parent" type="xs:string" />
      <xs:element name="guardian" type="xs:string" />
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Because the definition uses the `choice` compositor, the `sponsor` element can contain either element, but not both.

Finally, the following definition uses the `all` compositor to allow the `Family` element to contain elements named `Father` and/or `Mother`:

```
<xs:element name="Family">
  <xs:complexType>
    <xs:all>
      <xs:element name="Father" type="xs:string" />
      <xs:element name="Mother" type="xs:string" />
    </xs:all>
  </xs:complexType>
</xs:element>
```

It is also acceptable for the `Family` element to contain neither a `Father` nor a `Mother` element.

**TIP**

A complex element can contain only one all compositor; you cannot combine the all compositor with the choice or sequence compositor.

The choice and sequence compositors can be nested and combined. For example, the following definition uses two choice compositors nested within a sequence compositor to require the Account element to contain either the Person or the Company element followed by either the Cash or the Credit element:

```
<xs:element name="Account">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element name="Person" type="xs:string" />
        <xs:element name="Company" type="xs:string" />
      </xs:choice>
      <xs:choice>
        <xs:element name="Cash" type="xs:string" />
        <xs:element name="Credit" type="xs:string" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## Defining an Element Containing Nested Elements and Attributes

The next complex element you'll consider is an element containing both child elements and attributes. To define an element with this kind of content, you use the structure

```
<xs:element name="name">
  <xs:complexType>
    <xs:compositor>
      elements
    </xs:compositor>
    attributes
  </xs:complexType>
</xs:element>
```

where *name* is the name of the element; *compositor* is either *sequence*, *choice*, or *all*; *elements* represents a list of child elements nested within the element; and *attributes* represents a list of attribute definitions associated with the element. This is the same structure used for elements containing nested children except that a list of attributes is included. For example, the *student* element from Sabrina's *students.xml* file contains two attributes (*stuID* and *degree*) and six child elements (*lastName*, *firstName*, *dateOfEnrollment*, *comment*, *credits*, and *gpa*). Because you've already defined the content for the *stuID* attribute and the six child elements, you can insert references to those earlier definitions in the code, as follows:

```
<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="lastName" />
      <xs:element ref="firstName" />
      <xs:element ref="dateOfEnrollment" />
      <xs:element ref="credits" />
      <xs:element ref="comment" />
      <xs:element ref="gpa" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

<xs:attribute ref="stuID" />
</xs:complexType>
</xs:element>

```

You'll add this definition of the `student` element to the `students.xsd` file now.

### To define the student element in the students.xsd file:

- 1. Below the definition of the `gpa` element, insert the following code:

```

<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="lastName" />
      <xs:element ref="firstName" />
      <xs:element ref="dateOfEnrollment" />
      <xs:element ref="credits" />
      <xs:element ref="comment" />
      <xs:element ref="gpa" />
    </xs:sequence>

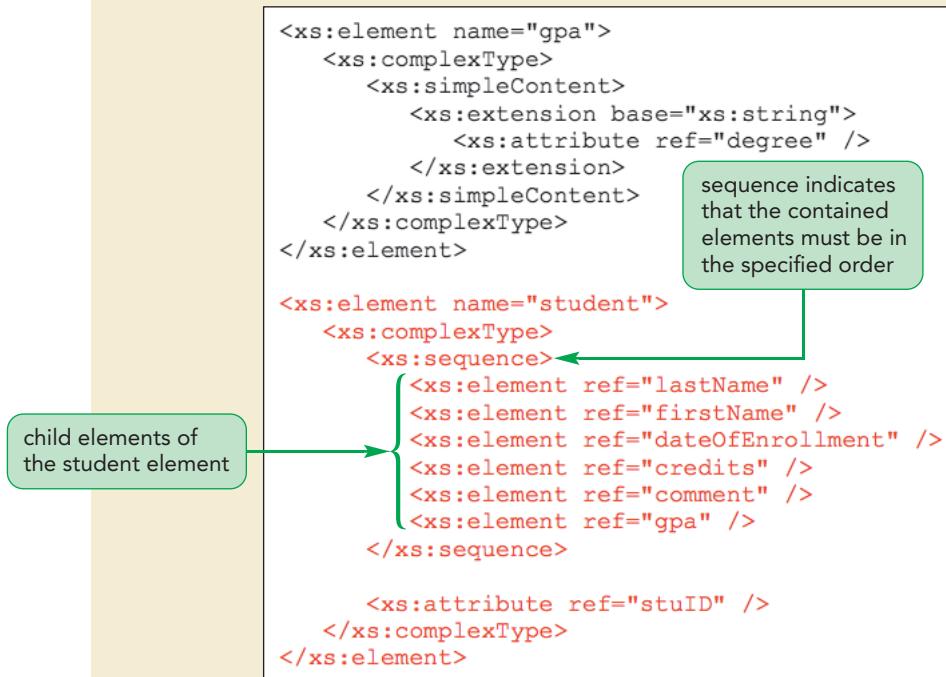
    <xs:attribute ref="stuID" />
  </xs:complexType>
</xs:element>

```

Figure 3-11 shows the code for the `student` element in the `students.xsd` file.

Figure 3-11

### Element containing both child elements and attributes



- 2. Save your changes to the file.

The only element from the students.xml file you haven't yet declared is the root `students` element. This element has no attributes but does contain the `student` element as a child. The definition of this element references the definition of the `student` element that you created earlier:

```
<xs:element name="students">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="student" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

You'll add this definition to the file now.

### To define the `students` element in the `students.xsd` file:

- 1. Below the definition of the `student` element, insert the following code:

```
<xs:element name="students">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="student" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 3-12 shows the code for the `students` element in the `students.xsd` file.

**Figure 3-12**

### Element containing only a child element

the only child element of  
the root `students` element  
is the `student` element

```
<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="lastName" />
      <xs:element ref="firstName" />
      <xs:element ref="dateOfEnrollment" />
      <xs:element ref="credits" />
      <xs:element ref="comment" />
      <xs:element ref="gpa" />
    </xs:sequence>

    <xs:attribute ref="stuID" />
  </xs:complexType>
</xs:element>

<xs:element name="students">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="student" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- 2. Save your changes to the file.

**INSIGHT**

### Specifying Mixed Content

One limitation of using DTDs is their inability to define mixed content. An element is said to have **mixed content** when it contains both a text string and child elements. You can specify the child elements with a DTD, but you cannot constrain their order or number. XML Schema gives you more control over mixed content. To specify that an element contains both text and child elements, you add the `mixed` attribute to the `<complexType>` tag. When the `mixed` attribute is set to the value `true`, XML Schema assumes that the element contains both text and child elements. The structure of the child elements can then be defined with the conventional method. For example, assume you were working on a document containing the following XML content:

```
<summary>
    Student <firstName>Cynthia</firstName>
    <lastName>Berstein</lastName> is enrolled in an IT
    degree program and has completed <credits>12</credits>
    credits since 01/01/2017.
</summary>
```

You could declare the `summary` element for this document in a schema file using the following complex type:

```
<element name="summary">
    <complexType mixed="true">
        <sequence>
            <element name="firstName" type="string" />
            <element name="lastName" type="string" />
            <element name="credits" type="string" />
        </sequence>
    </complexType>
</element>
```

In an element with mixed content, XML Schema allows text content to appear before, between, and after any child element.

At this point, you have defined all of the simple and complex types in Sabrina's document. Next, you'll refine your schema by adding code that defines exactly how these simple and complex types are used and what kind of data they can contain.

## Indicating Required Attributes

An attribute may or may not be required with a particular element. To indicate whether an attribute is required, the `use` attribute can be added to the statement that assigns the attribute to an element. The general syntax of the `use` attribute is

```
<xss:element name="name">
    <xss:complexType>
        element content
        <xss:attribute properties use="use" />
    </xss:complexType>
</xss:element>
```

where `use` is one of the following three values:

- **required**—The attribute must always appear with the element.
- **optional**—The use of the attribute is optional with the element.
- **prohibited**—The attribute cannot be used with the element.

For example, in Sabrina's document, the `degree` attribute is required with every `gpa` element. To force the instance document to follow this rule, you add the `use` attribute to the definition of the `degree` attribute, as follows:

```
<xs:attribute name="degree" type="xs:string" use="required" />
```

If you neglect to add the `use` attribute to an element declaration, XML parsers assume that the attribute is optional. The `use` attribute is applied only when assigning an attribute to a specific element from the instance document. After all, an attribute might be required for one element and optional for another.

The two attributes in Sabrina's document—`stuID` and `degree`—are both required for the document to be valid. You will indicate this in the schema by specifying the use of each attribute.

### To indicate required attributes in the students.xsd file:

- 1. Within the code that defines the `gpa` element, locate the `xs:attribute` tag for the `degree` attribute, and then add the code `use="required"` to the tag. This code indicates that the `degree` attribute is required for the `gpa` element.
- 2. Within the code that defines the `student` element, locate the `xs:attribute` tag for the `stuID` attribute, and then add the code `use="required"` to the tag. This code indicates that the `stuID` attribute is required for the `student` element. Figure 3-13 highlights the revised code in the schema.

Figure 3-13

Attributes designated as required

```
<xs:element name="gpa">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="degree" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="lastName" />
      <xs:element ref="firstName" />
      <xs:element ref="dateOfEnrollment" />
      <xs:element ref="credits" />
      <xs:element ref="comment" />
      <xs:element ref="gpa" />
    </xs:sequence>
    <xs:attribute ref="stuID" use="required" />
  </xs:complexType>
</xs:element>
```

the student element requires the use of the stuID attribute

the gpa element requires the use of the degree attribute

- 3. Save your changes to the file.

## Specifying the Number of Child Elements

The previous code samples assumed that each element in the list appeared once and only once. This is not always the case. For example, Sabrina's document contains information on one or more students, so you need to allow for one or more `student` elements. To specify the number of times an element appears in the instance document, you can apply the `minOccurs` and `maxOccurs` attributes to the element definition, using the syntax

```
<xs:element name="name" type="type" minOccurs="value"
            maxOccurs="value" />
```

where the value of the `minOccurs` attribute defines the minimum number of times the element can occur, and the value of the `maxOccurs` attribute defines the maximum number of times the element can occur. For example, the following element declaration specifies that the `student` element must appear at least once and may appear no more than three times in the instance document:

```
<xs:element name="student" type="xs:string" minOccurs="1"
            maxOccurs="3" />
```

### TIP

A `minOccurs` value of 0 with a `maxOccurs` value of unbounded is equivalent to the `*` character in a DTD. Likewise, values of 1 and unbounded are equivalent to the `+` character, and values of 0 and 1 are equivalent to the `?` character.

Any time the `minOccurs` attribute is set to 0, an element is optional. The `maxOccurs` attribute can be any positive value, or it can have a value of `unbounded` for unlimited occurrences of the element. If a value is specified for the `minOccurs` attribute but the `maxOccurs` attribute is missing, the value of the `maxOccurs` attribute is assumed to be equal to the value of the `minOccurs` attribute. Finally, if both the `minOccurs` attribute and the `maxOccurs` attribute are missing, the element is assumed to occur only once.

The `student` element occurs one or more times in Sabrina's document, and the `comment` element occurs zero or more times. All the other elements occur only once. You'll add the appropriate `minOccurs` and `maxOccurs` values to the schema now for the `student` and `comment` elements.

### To set the occurrences of the `student` and `comment` elements in the `students.xsd` file:

- 1. In the `students.xsd` file, within the code that defines the `student` element, locate the `xs:element` tag that references the `comment` element, and then add the code `minOccurs="0" maxOccurs="unbounded"` to the tag. This code allows for zero or more occurrences of the `comment` element.
- 2. Within the code that defines the `students` element, locate the `xs:element` tag that references the `student` element and then add the code `minOccurs="1" maxOccurs="unbounded"` to the tag. This code allows for one or more occurrences of the `student` element. Figure 3-14 highlights the new code in the schema.

Be sure to add the `minOccurs` and `maxOccurs` attributes to the `comment` element reference within the definition for the `student` element, and *not* within the definition of the `comment` element at the start of the schema.

Figure 3-14

## The minOccurs and maxOccurs values

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:attribute name="stuID" type="xs:string" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="dateOfEnrollment" type="xs:string" />
    <xs:element name="credits" type="xs:string" />
    <xs:element name="comment" type="xs:string" />
    <xs:attribute name="degree" type="xs:string" />

    <xs:element name="gpa">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:string">
                    <xs:attribute ref="degree" use="required" />
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>

    <xs:element name="student">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="lastName" />
                <xs:element ref="firstName" />
                <xs:element ref="dateOfEnrollment" />
                <xs:element ref="credits" />
                <xs:element ref="comment" minOccurs="0" maxOccurs="unbounded" />
                <xs:element ref="gpa" />
            </xs:sequence>
        <xs:complexType>
    </xs:element>

    <xs:element name="students">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="student" minOccurs="1" maxOccurs="unbounded" />
            </xs:sequence>
        <xs:complexType>
    </xs:element>

</xs:schema>

```

- 3. Save your changes to the file.

## Validating a Schema Document

You're ready to test whether the `students.xsd` document validates. The web has many excellent sources for validating parsers including websites such as the W3C's, where you can upload or paste schema code to have it validated. Several editors provide schema validation as well.

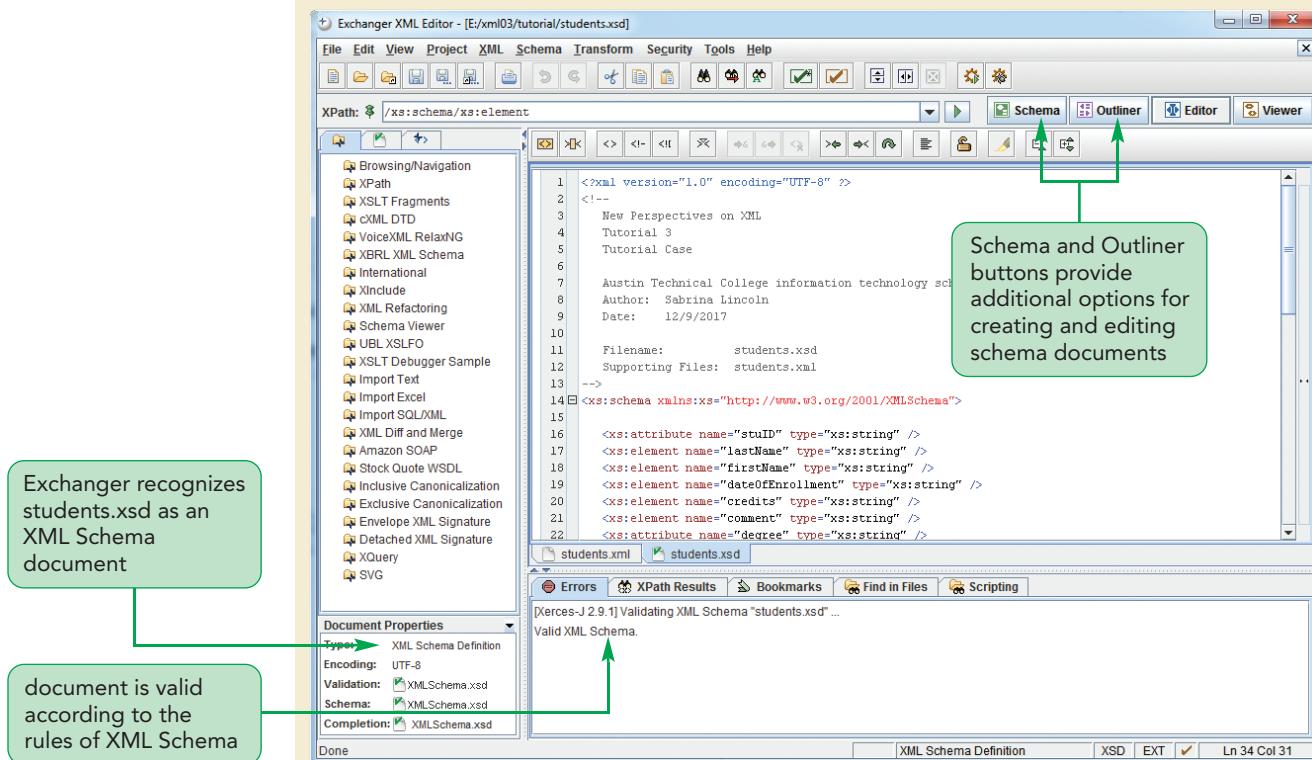
The following steps use Exchanger XML Editor to validate Sabrina's `students.xsd` document.

### To validate the students.xsd file:

- 1. If necessary, open **students.xsd** in Exchanger XML Editor.
- 2. Click **Schema** on the Menu bar, and then click **Validate XML Schema**. As shown in Figure 3-15, the error console reports that the students.xsd file is valid.

Figure 3-15

### XML Schema validation



**Trouble?** If a validation error is reported in the Errors tab, there is an error in the schema. Check your schema code against the students.xsd code shown in Figure 3-14. Your code should match exactly, including the use of uppercase and lowercase letters. Fix any discrepancies, be sure to save your changes, validate the schema again, and then reexamine the Errors tab for validation information.

**Trouble?** If your Exchanger XML Editor window doesn't match the one shown in Figure 3-15, close students.xsd and reopen it.

Now that you've confirmed that your schema is valid, you can apply it to your XML document.

## Applying a Schema to an Instance Document

To attach a schema to an instance document, you declare the XML Schema instance namespace in the instance document, and then you specify the location of the schema file. To declare the XML Schema instance namespace, you add the following attribute to the root element of the instance document:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Although the prefix `xsi` is commonly used for the XML Schema Instance namespace, you can specify a different prefix in your documents.

You add a second attribute to the root element to specify the location of the schema file. The attribute you use depends on whether the instance document is associated with a namespace. If the document is not associated with a namespace, you add the attribute

```
xsi:noNamespaceSchemaLocation="schema"
```

to the root element, where `schema` is the location and name of the schema file. Note that the attribute requires the `xsi` namespace prefix because the attribute itself is from the XML Schema Instance namespace.



### Teamwork: Working with Multiple Schema Documents

Especially on a large project involving multiple programmers, it can be useful to create multiple schema documents rather than a single large schema document. You can use the `xs:include` element within an XML Schema document to include the contents of another schema document. For instance, a project might break up into two teams—one creating `customer.xsd`, a schema for customer content, and another creating `products.xsd`, a schema for order information. To validate against the contents of both of these schemas, you would create a master schema file for your project and add the following two lines of code to it:

```
<xs:include schemaLocation="customer.xsd" />
<xs:include schemaLocation="products.xsd" />
```

Using the `xs:include` element enables you to validate instance documents against the validation rules defined in multiple XML Schema documents.

Sabrina has not yet placed the contents of her `students.xml` document in a namespace, so you'll add the following attribute to the root `students` element:

```
xsi:noNamespaceSchemaLocation="students.xsd"
```

#### To apply the `students.xsd` schema to the `students.xml` document:

- ➊ Return to the `students.xml` file in your editor.
- ➋ Within the opening tag for the `students` element, add the following attributes:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="students.xsd"
```

Figure 3-16 shows the namespace attributes inserted in the root element.

Figure 3-16

### Schema applied to a document without a namespace

XML Schema instance namespace

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="students.xsd">
    <student stuID="SI890-041-02">
        <lastName>Berstein</lastName>
        <firstName>Cynthia</firstName>
        <dateOfEnrollment>2017-05-22</dateOfEnrollment>
        <credits>12</credits>
        <gpa degree="MP">3.81</gpa>
    </student>
```

name of the schema document

- 3. Save your changes to the file.

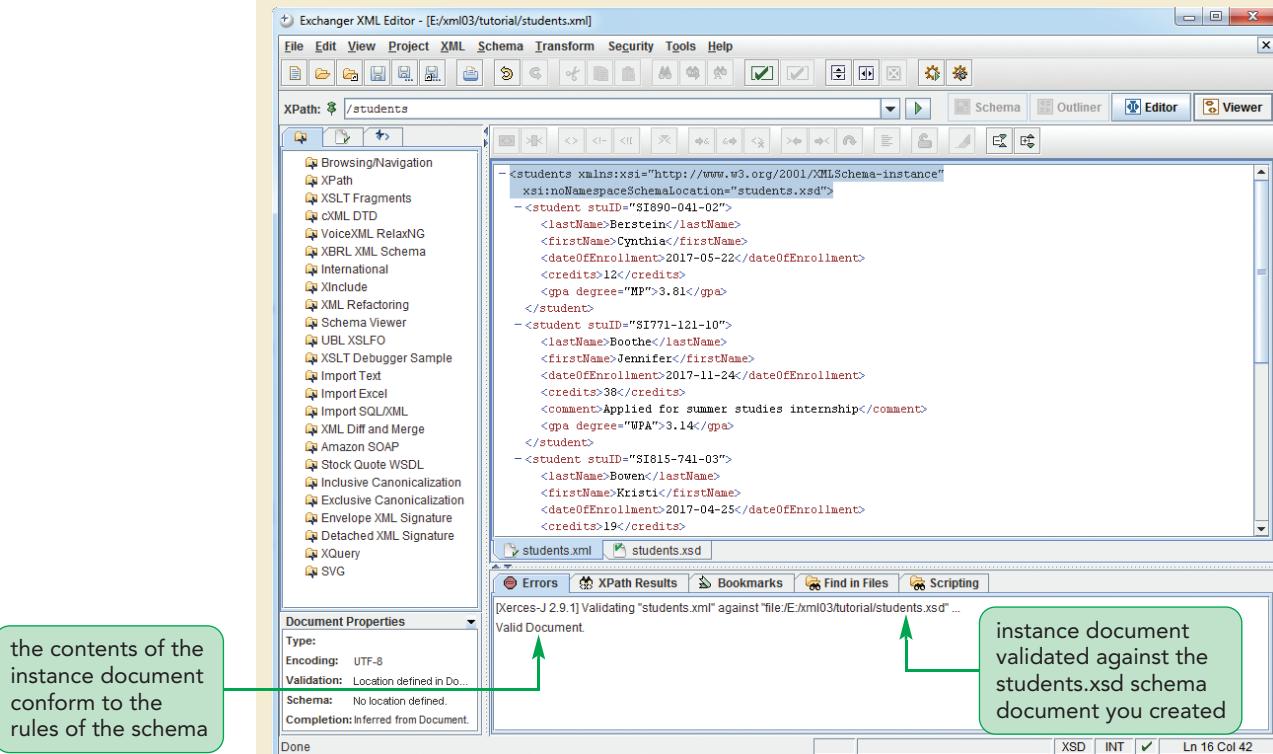
Now that you've applied your schema to Sabrina's instance document, you can validate the document against the rules defined in the students.xsd file. To validate the document, you use an XML parser. Although the following steps use the validator within Exchanger XML Editor, you can also validate using one of the free or commercial XML validating parsers available on the web.

### To validate the students.xml document:

- 1. If necessary, open **students.xml** in Exchanger XML Editor.
- 2. Click the **Viewer** button in the upper-right corner of the window.
- 3. Click **XML** on the menu bar, and then click **Validate**. As shown in Figure 3-17, the error console reports that the students.xml file is a valid document.

Figure 3-17

### Validation results for the students.xml file



**Trouble?** If a validation error is reported in the Errors tab and your schema validated successfully in the earlier steps, then there is an error in the namespace attributes you entered in the instance document. Check your students.xml code against the namespace attributes shown in Figure 3-16. Your code should match exactly, including the use of uppercase and lowercase letters. Fix any discrepancies, save your changes, and then revalidate.

Sabrina suggests that you add an intentional error to the students.xml file to confirm that the document is rejected as invalid. To do this, you'll add a second `credits` element to the first student's data. Because the schema you wrote permits only one `credits` element per student, this should result in an invalid document.

### To add an error to the students.xml file:

1. Return to the `students.xml` file in Exchanger XML Editor, and then click the **Editor** button in the upper-right corner.
2. Within the `student` element containing information on Cynthia Berstein, add the following code:

```
<credits>14</credits>
```

Figure 3-18 shows the erroneous element added to the students.xml file.

Figure 3-18

### Second credits element added to the students.xml file

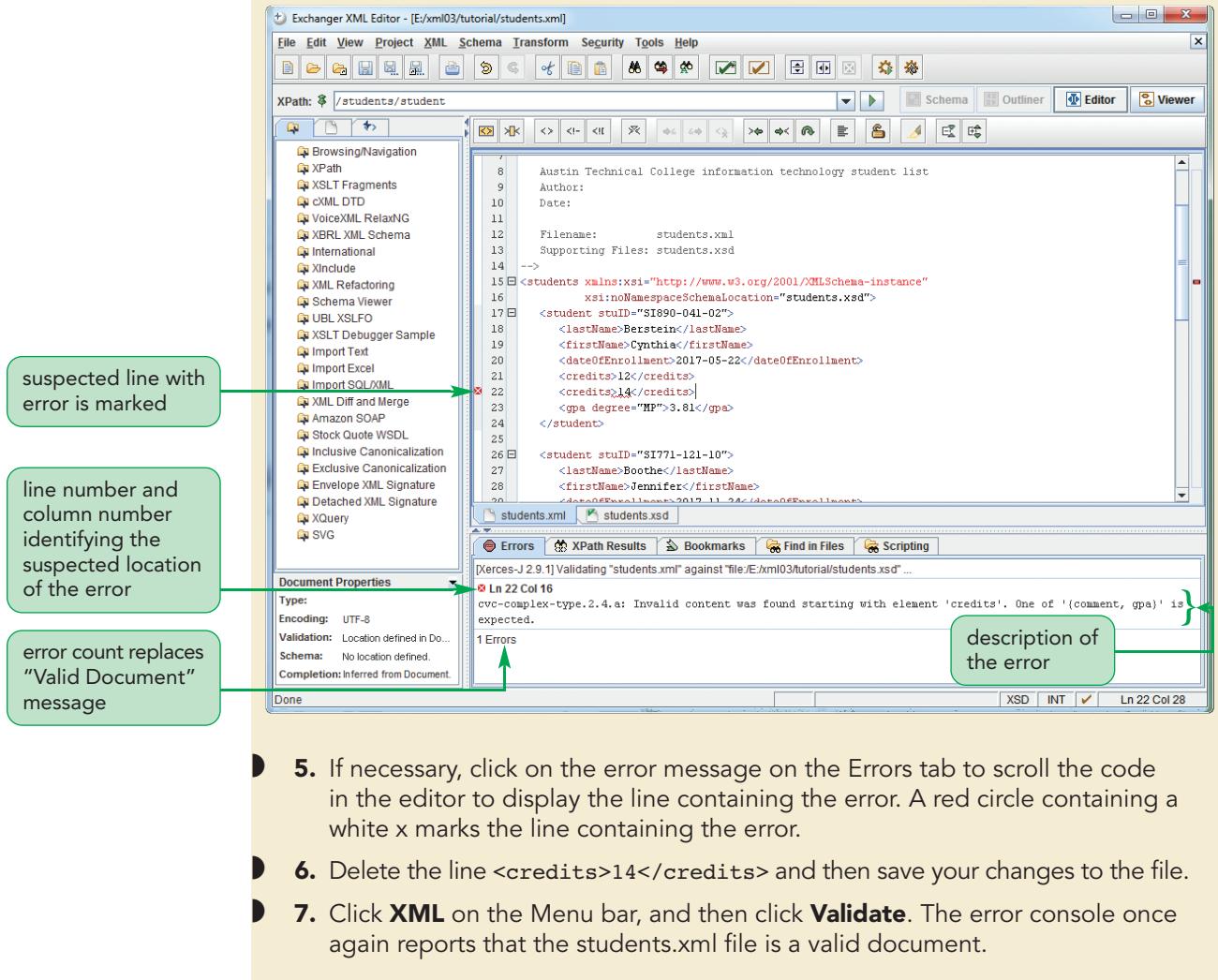
a second credits element is not valid according to the schema

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="students.xsd">
    <student stuID="SI890-041-02">
        <lastName>Berstein</lastName>
        <firstName>Cynthia</firstName>
        <dateOfEnrollment>2017-05-22</dateOfEnrollment>
        <credits>12</credits>
        <credits>14</credits>
        <gpa degree="MP">3.81</gpa>
    </student>
```

3. Save your changes to the file.
4. Click **XML** on the Menu bar, and then click **Validate**. The document is rejected as invalid due to the extra `credits` element. See Figure 3-19.

Figure 3-19

## Validation error



Sabrina is pleased with the initial work you've done on designing a schema for the students.xml document. In the next session, you'll implement additional validation rules that Sabrina would like applied to her document as you learn about data types.

**REVIEW****Session 3.1 Quick Check**

1. What is a schema? What is an instance document?
2. How do schemas differ from DTDs?
3. What is a simple type? What is a complex type?
4. How do you declare a simple type element named `Address` that contains string data?
5. How do you declare a complex type element named `Address` that contains, in order, the child elements `Apartment` (optional), `city`, `State`, and `zip`? (Assume that the `Apartment`, `City`, `State`, and `zip` elements are simple type elements containing text strings.)
6. The `Book` element contains simple text and a `title` attribute. What code would you enter into a schema file to define this complex type element?
7. What code would you enter into a schema to create a reference to an attribute named `studentID`?
8. What attributes would you add to the root element of an instance document to attach it to a schema file named `schema1.xsd`? Assume that no namespace has been assigned to the schema file, and that you're using the XML Schema vocabulary.

## Session 3.2 Visual Overview:

The `siType` data type is a **user-derived data type**, which is a data type defined by a schema's author.

A regular expression is a text string that defines a character pattern.

A **pattern** is a constraining facet that limits data to a general pattern.

```

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:attribute name="stuID" type="siType" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="dateOfEnrollment" type="xs:date" />
    <xs:element name="credits" type="creditsType" />
    <xs:element name="comment" type="xs:string" />
    <xs:attribute name="degree" type="degreeType" />

    <xs:simpleType name="siType">
        <xs:restriction base="xs:ID">
            <xs:pattern value="SI\d{3}-\d{3}-\d{2}" />
        </xs:restriction>
    </xs:simpleType>

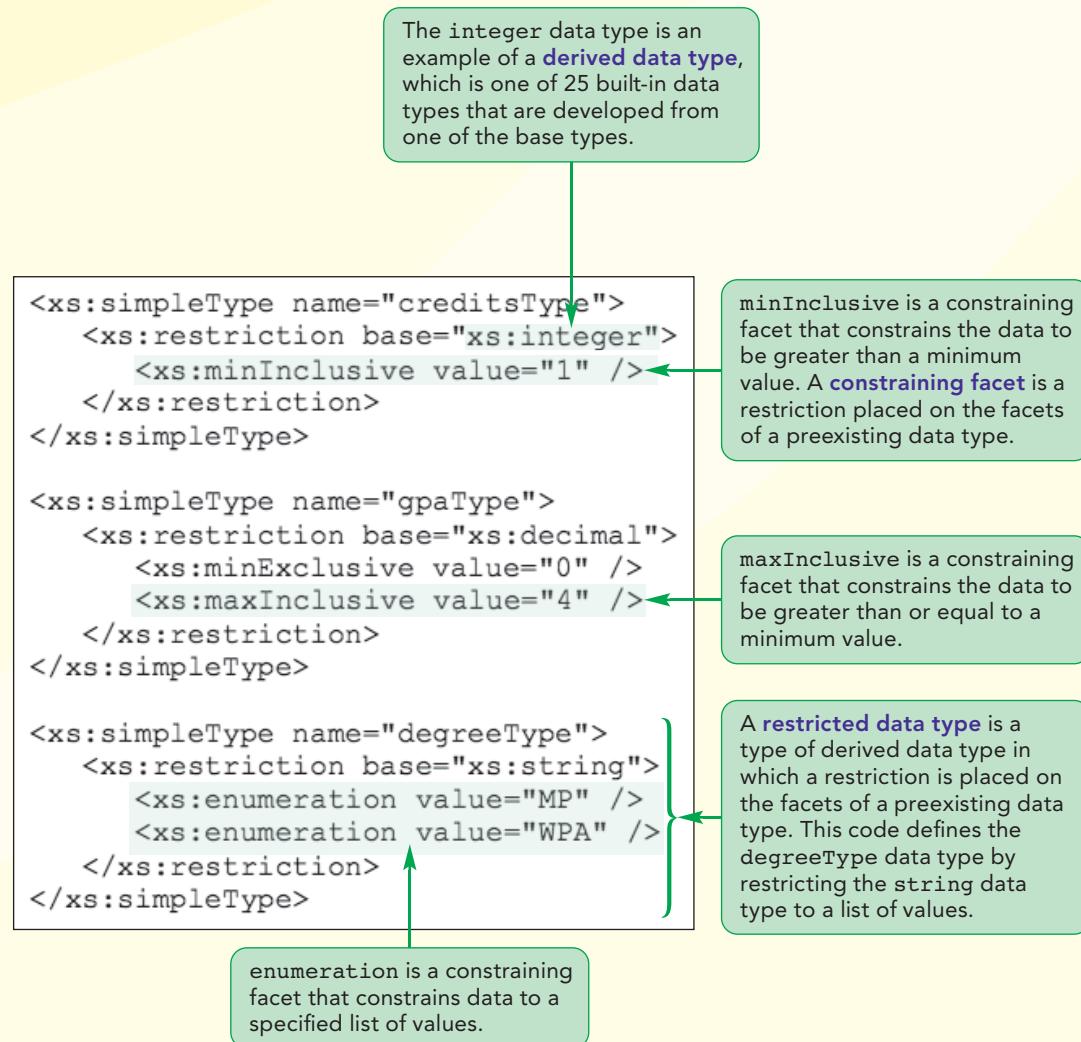
    <xs:element name="gpa">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="gpaType">
                    <xs:attribute ref="degree" use="required" />
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>

```

The string data type is a **built-in data type**, which is part of the XML Schema language. The string data type is an example of a **primitive data type** (also called a **base type**), which is a subgroup of built-in data types that are not defined in terms of other types.

A **character type** is a representation of a specific type of character. For instance, `\d` is the character type for a single digit.

# Validating with Data Types



## Validating with Built-In Data Types

The schema you designed for the students.xml document uses the `string` data type for all element and attribute content, which allows users to enter any text string into those items. Sabrina wants to ensure that dates are entered in the proper form, that only positive integers are entered for the student credits, and that the student IDs follow a prescribed pattern. You can do all of these using additional data types supported by XML Schema.

### TIP

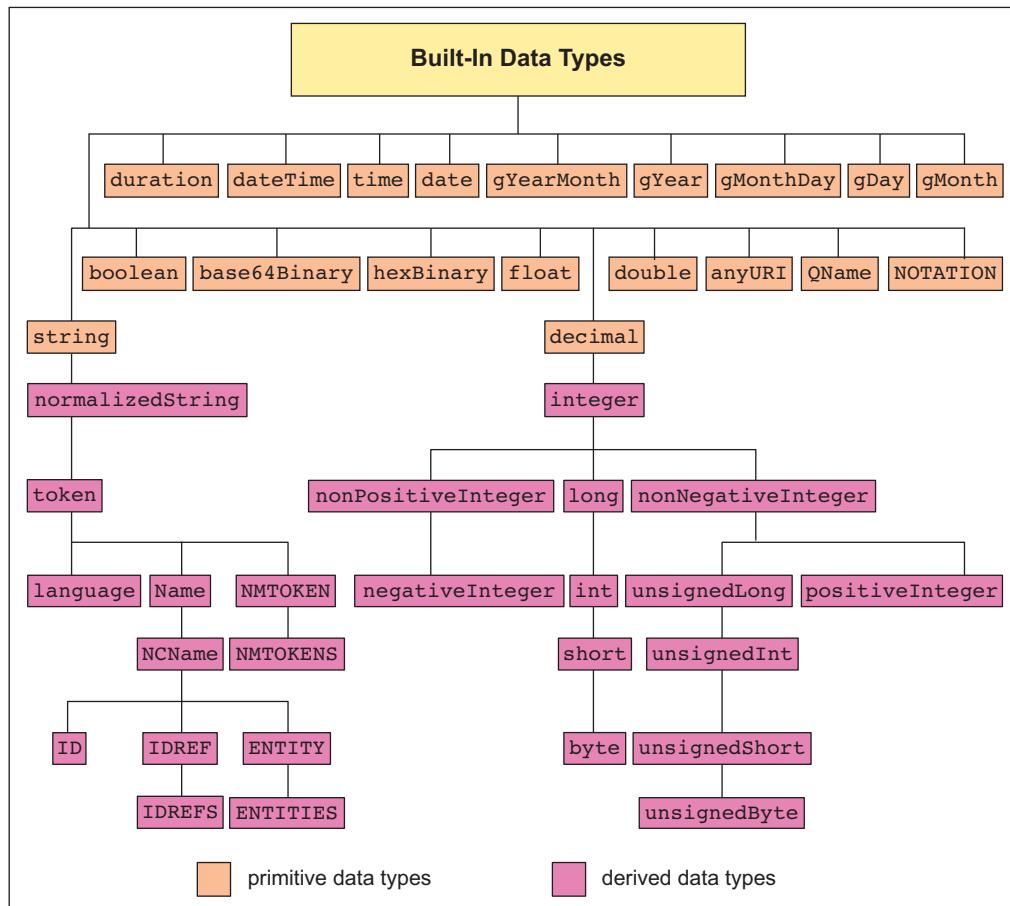
All built-in data types are part of the XML Schema vocabulary and must be placed in the XML Schema namespace.

XML Schema supports two general categories of data types—built-in and user-derived. A built-in data type is part of the XML Schema language. A user-derived data type is a data type defined by a schema's author. You'll begin your work with data types by exploring the built-in data types in XML Schema.

XML Schema divides its built-in data types into two classes—primitive and derived. A primitive data type, also called a base type, is one of 19 fundamental data types that are not defined in terms of other types. A **derived data type** is one of 25 data types that are developed from one of the base types. Figure 3-20 provides a schematic diagram of all 44 built-in data types.

Figure 3-20

XML Schema built-in data types



Derived data types share many of the same characteristics as the primitive data types from which they are derived, but incorporate one or two additional restrictions or modifications. To see how this is done, you'll examine the `string` data types.

## String Data Types

In the previous session, you used only the primitive `string` data type, allowing almost any text string in the elements and attributes of Sabrina's document. The `string` data type is the most general of XML Schema's built-in data types. For that reason, it is not very useful if you need to exert more control over element and attribute values in an instance document. XML Schema provides several derived data types that enable you to restrict text strings. Figure 3-21 describes some of these data types.

Figure 3-21

Some data types derived from `string`

Data Type	Description
<code>xs:string</code>	A text string containing all legal characters from the ISO/IEC character set, including all white space characters
<code>xs:normalizedString</code>	A text string in which all white space characters are replaced with blank spaces
<code>xs:token</code>	A text string in which adjoining blank spaces are replaced with a single blank space, and opening and closing spaces are removed
<code>xs:NMTOKEN</code>	A text string containing valid XML names with no white space
<code>xs:NMTOKENS</code>	A list of NMTOKEN data values separated by white space
<code>xs: Name</code>	A text string similar to the NMTOKEN data type except that names must begin with a letter or the colon (:) or hyphen (-) character
<code>xs:NCName</code>	A "noncolonized name," derived from the Name data type but restricting the use of colons anywhere in the name
<code>xs:ID</code>	A unique ID name found nowhere else in the instance document
<code>xs:IDREF</code>	A reference to an ID value found in the instance document
<code>xs:IDREFS</code>	A list of ID references separated by white space
<code>xs:ENTITY</code>	A value matching an unparsed entity defined in a DTD
<code>xs:ENTITIES</code>	A list of entity values matching unparsed entities defined in a DTD

Some of the data types in the list should be familiar from your work with DTDs. For example, the `ID` data type allows text strings containing unique ID values, and the `IDREF` and `IDREFS` data types allow only text strings that contain references to ID values located in the instance document.

### REFERENCE

#### Applying Built-In XML Schema Data Types

- For any string content, use the data type `xs:string`.
- For an ID value, use `xs:ID`.
- For a reference to an ID value, use `xs:IDREF`.
- For a decimal value, use `xs:decimal`.
- For an integer value, use `xs:integer`.
- For a positive integer, use `xs:positiveInteger`.
- For a date in the format `yyyy-mm-dd`, use `xs:date`.
- For a time in the format `hh:mm:ss`, use `xs:time`.

Each student in Sabrina's document has a `stuID` attribute that uniquely identifies the student. You'll apply the `ID` data type to this attribute now.

### To apply the ID data type:

- 1. If you took a break after the previous session, make sure the **students.xml** and **students.xsd** files are open in your editor.
- 2. Within the **students.xsd** file, locate the definition for the **stuID** attribute.
- 3. Change the **type** value from **xs:string** to **xs:ID**. Just like the **string** data type, the **ID** data type is built into XML Schema, so you identify it using the **xs** namespace prefix. Figure 3-22 shows the updated **type** value.

Figure 3-22

### ID data type applied

**stuID values must be unique IDs**

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:attribute name="stuID" type="xs:ID" />
  <xs:element name="lastName" type="xs:string" />
```

► 4. Save your changes to the **students.xsd** file.

## Numeric Data Types

Unlike DTDs, schemas that use XML Schema support numeric data types. Most numeric data types are derived from four primitive data types—**decimal**, **float**, **double**, and **boolean**. Figure 3-23 describes these and some other XML Schema numeric data types.

Figure 3-23

### Numeric data types

Data Type	Description
<b>xs:decimal</b>	A decimal number in which the decimal separator is always a dot (.) with a leading + or - character allowed; no nonnumeric characters are allowed, nor is exponential notation
<b>xs:integer</b>	An integer
<b>xs:nonPositiveInteger</b>	An integer less than or equal to zero
<b>xs:negativeInteger</b>	An integer less than zero
<b>xs:nonNegativeInteger</b>	An integer greater than or equal to zero
<b>xs:positiveInteger</b>	An integer greater than zero
<b>xs:float</b>	A floating point number allowing decimal values and values in scientific notation; infinite values can be represented by -INF and INF; nonnumeric values can be represented by NaN
<b>xs:double</b>	A double precision floating point number
<b>xs:boolean</b>	A Boolean value that has the value <b>true</b> , <b>false</b> , <b>0</b> , or <b>1</b>

Sabrina's XML document includes the total credits for each of the students in the information technology programs. She wants you to validate that all the values she entered for the `credits` element are positive integers. Sabrina also entered a numeric score for each student's GPA. The GPA values range from 0 to 4. She wants you to change the data type for the `gpa` element to `decimal`. Because `gpa` is a complex type element containing text and attributes, you'll add the data type to the `base` attribute in the `simpleContent` element. You'll make these changes to your schema document now.

### To apply the positiveInteger and decimal data types:

1. Within the `students.xsd` file, locate the `xs:element` tag for the `credits` element and then change the value of the `type` attribute from `xs:string` to `xs:positiveInteger`.
2. Within the code to define the `gpa` element, locate the opening `xs:extension` tag and then change the value of the `base` attribute from `xs:string` to `xs:decimal`. Figure 3-24 shows the revised code.

**Figure 3-24** positiveInteger and decimal data types applied

values for the gpa element are limited to decimal values

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="stuID" type="xs:ID" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="dateOfEnrollment" type="xs:string" />
  <xs:element name="credits" type="xs:positiveInteger" />
  <xs:element name="comment" type="xs:string" />
  <xs:attribute name="degree" type="xs:string" />

  <xs:element name="gpa">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:decimal">
          <xs:attribute ref="degree" use="required" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
```

values for the credits element are limited to positive integers

3. Save your changes to the `students.xsd` file.

## Data Types for Dates and Times

### TIP

To support date strings such as 1/8/2017 or Jan. 8, 2017, you must create your own date type or use a date type library created by another XML developer.

XML Schema provides several data types for dates, times, and durations. However, XML Schema does not allow for any flexibility in the date and time formats it uses. For instance, date values containing a month, day, and year must be entered in the format

`yyyy-mm-dd`

where `yyyy` is the four-digit year value, `mm` is the two-digit month value, and `dd` is the two-digit day value. Month values range from 01 to 12, and day values range from 01 to 31 (depending on the month). The date value

2017-01-08

would be valid under XML Schema, but the date value

2017-1-8

would not be valid because its month and day values are not two-digit integers.

Times in XML Schema must be entered using 24-hour (or military) time. The format is

*hh:mm:ss*

where *hh* is the hour value ranging from 00 to 23, and *mm* and *ss* are the minutes and seconds values, respectively, ranging from 00 to 59. No data type exists in XML Schema for expressing time in the 12-hour AM/PM format. In the *time* data type, each time value (hours, minutes, and seconds) must be specified. Thus, the time value

15:45

would be invalid because it does not specify a value for seconds. Figure 3-25 summarizes the different data types supported by XML Schema for dates and times.

Figure 3-25

Date and time data types

Data Type	Description
<code>xs:dateTime</code>	A date and time entered in the format <i>yyyy-mm-ddT hh:mm:ss</i> where <i>yyyy</i> is the four-digit year, <i>mm</i> is the two-digit month, <i>dd</i> is the two-digit day, <i>T</i> is the time zone, <i>hh</i> is the two-digit hour, <i>mm</i> is the two-digit minute, and <i>ss</i> is the two-digit second
<code>xs:date</code>	A date entered in the format <i>yyyy-mm-dd</i>
<code>xs:time</code>	A time entered in the format <i>hh:mm:ss</i>
<code>xs:gYearMonthDay</code>	A date based on the Gregorian calendar entered in the format <i>yyyy-mm-dd</i> (equivalent to <code>xs:date</code> )
<code>xs:gYearMonth</code>	A date entered in the format <i>yyyy-mm</i> (no day is specified)
<code>xs:gYear</code>	A year entered in the format <i>yyyy</i>
<code>xs:gMonthDay</code>	A month and day entered in the format <i>--mm-dd</i>
<code>xs:gMonth</code>	A month entered in the format <i>--mm</i>
<code>xs:gDay</code>	A day entered in the format <i>--dd</i>
<code>xs:duration</code>	A time duration entered in the format <i>PyYmMdDhHmMsS</i> where <i>y</i> , <i>m</i> , <i>d</i> , <i>h</i> , <i>m</i> , and <i>s</i> are the duration values in years, months, days, hours, minutes, and seconds, respectively; an optional negative sign is also permitted to indicate a negative time duration

Sabrina recorded each student's date of enrollment in her XML document using the *dateOfEnrollment* element. You'll apply the *date* data type to values of this element to confirm that she entered all the date values correctly.

### To apply the date data type:

- 1. Within the **students.xsd** file, locate the `xs:element` tag for the *dateOfEnrollment* element, and then change the data type from `xs:string` to `xs:date`, as shown in Figure 3-26.

Figure 3-26

**date data type applied**

dateOfEnrollment  
values must be  
entered in the  
format yyyy-mm-dd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:attribute name="stuID" type="xs:ID" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="dateOfEnrollment" type="xs:date" />
  <xs:element name="credits" type="xs:positiveInteger" />
  <xs:element name="comment" type="xs:string" />
  <xs:attribute name="degree" type="xs:string" />
```

- 2. Save your changes to the **students.xsd** file.

Now that you've added more specific data types to the students.xsd file, you'll validate the contents of the students.xml file to verify that Sabrina's data matches the rules of the schema.

**To validate the students.xml file:**

- 1. In your XML editor, switch to the **students.xml** file and then click the **Viewer** button in the upper-right corner of the window.
- 2. On the Menu bar, click **XML** and then click **Validate**. The document validates successfully against the schema.

Sabrina suggests that you add a few errors to the students.xml file to confirm that the modifications you just made to the schema work as expected. The schema you wrote allows only integer values for **credits** element, and allows only decimal values for the **gpa** element. In addition, date values for the **dateOfEnrollment** must follow a strict pattern. You'll modify the **credits**, **gpa**, and **dateOfEnrollment** values for the first student's data so they don't satisfy these criteria; if the schema works as intended, these changes should result in an invalid document.

**To add errors to the students.xml file:**

- 1. Return to the **students.xml** file in your XML editor, and then click the **Editor** button in the upper-right corner of the window.
- 2. Within the **student** element for Cynthia Bernstein, in the **credits** element value, add a **space** after the number 12 and then type the word **credits**. With the additional characters, the value is no longer an integer.
- 3. Within the **gpa** element for the same student, delete the element value 3.81. The empty value does not match the element's **decimal** data type.
- 4. Within the **dateOfEnrollment** element for the same student, delete the leading 0 from the month value so the date reads **2017-5-22**. The date no longer matches the **yyyy-mm-dd** pattern for the **date** data type. Figure 3-27 shows the revised data for the first student in the document.

Figure 3-27

## Three errors introduced into instance document

```

<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="students.xsd">
    <student stuid="SI890-041-02">
        <lastName>Berstein</lastName>
        <firstName>Cynthia</firstName>
        <dateOfEnrollment>2017-5-22</dateOfEnrollment>
        <credits>12 credits</credits>
        <gpa degree="MP"></gpa>
    </student>

```

- ▶ 5. Save your changes to the **students.xml** file.
- ▶ 6. On the Menu bar, click **XML**, click **Validate**, and then scroll through the list of errors on the Errors tab. The document is rejected with errors found due to the **credits**, **gpa**, and **dateOfEnrollment** element values you edited no longer matching the schema definitions.
- ▶ 7. In the **credits** element for the first record, delete the space and the word "credits" that you entered so the value is once again 12; in the **gpa** element for the first record, enter the value **3.81**; in the **dateOfEnrollment** element, change the value to read **2017-05-22**; and then save your changes to the file.
- ▶ 8. Revalidate the document and confirm that it passes the validation test.
- ▶ 9. Save your changes to the **students.xml** file.

## Deriving Customized Data Types

In addition to the built-in data types you’re using, you also need to create some new data types to fully validate Sabrina’s document. Sabrina has provided the following additional rules for the elements and attributes in her document:

- The value of the **credits** element must be at least 1.
- The **gpa** value must fall between 0 and 4.
- The value of the **degree** attribute must be either **MP** or **WPA**. (The value **MP** stands for Mobile Programmer, and the value **WPA** stands for Web Programmer/Analyst.)

Although XML Schema has no built-in data types for these rules, you can use it to derive—or build—your own data types. The code to derive a new data type is

```

<xs:simpleType name="name">
    rules
</xs:simpleType>

```

where *name* is the name of the user-defined data type and *rules* is the list of statements that define the properties of that data type. This structure is also known as a named simple type because it defines simple type content under a name provided by the schema author. You can also create a simple type without a name, which is known as an **anonymous simple type**. The code for an anonymous simple type is easier to create. However, one advantage of creating a named simple type is that you can reference that simple type elsewhere in your schema using the simple type name.

**TIP**

Just as you can create customized simple types, you can also create customized complex types, which allow you to reuse complex structures in a schema document.

Each new data type must be derived from a preexisting data type found in either XML Schema or a user-defined vocabulary. The following three components are involved in deriving any new data type:

- **value space**—The set of values that correspond to the data type. For example, the value space for a `positiveInteger` data type includes the numbers 1, 2, 3, etc., but not 0, negative integers, fractions, or text strings.
- **lexical space**—The set of textual representations of the value space. For example, a value supported by the `floating` data type, such as 42, can be represented in several ways, including 42, 42.0, or 4.2E01.
- **facets**—The properties that distinguish one data type from another. Facets can include such properties as text string length or a range of allowable values. For example, a facet that distinguishes the `integer` data type from the `positiveInteger` data type is the fact that positive integers are constrained to the realm of positive numbers.

New data types are created by manipulating the properties of these three components. You can do this by:

1. Creating a list based on preexisting data types
2. Creating a union of one or more of the preexisting data types
3. Restricting the values of a preexisting data type



### PRO SKILLS Problem Solving: Reusing Code with Named Model Groups and Named Attribute Groups

Named types are not the only structures you can create to be reused in your schemas. Another structure is a named model group. As the name suggests, a **named model group** is a collection, or group, of elements. The syntax for creating a model group is

```
<xs:group name="name">
    elements
</xs:group>
```

where `name` is the name of the model group and `elements` is a collection of element declarations. Model groups are useful when a document contains element declarations or code that you want to repeat throughout the schema.

Like elements, attributes can be grouped into collections called **named attribute groups**. This is particularly useful for attributes that you want to use with several different elements in a schema. The syntax for a named attribute group is

```
<xs:attributeGroup name="name">
    attributes
</xs:attributeGroup>
```

where `name` is the name of the attribute group and `attributes` is a collection of attributes assigned to the group.

You'll start by examining how to create a list data type.

## Deriving a List Data Type

A **list data type** is a list of values separated by white space, in which each item in the list is derived from an established data type. You already have seen a couple of examples of list data types found in XML Schema, including the `xs:ENTITIES` and `xs:IDREFS` lists. In these cases, the list data types are derived from XML Schema's `xs:ENTITY` and `xs:IDREF` data types. The syntax for deriving a customized list data type is

```
<xs:simpleType name="name">
  <xs:list itemType="type" />
</xs:simpleType>
```

### TIP

A list data type must always use white space as the delimiter. You cannot use commas or other non-white space characters.

where `name` is the name assigned to the list data type and `type` is the data type from which each item in the list is derived. For example, Austin Technical College might decide to include a student's GPA for each semester along with the student's overall GPA. An element containing the GPA information by semester might appear as follows:

```
<semGPA>3.81 3.92 3.3 3.2</semGPA>
```

To create a data type for this information, you could define the following named simple type:

```
<xs:simpleType name="semList">
  <xs:list itemType="xs:decimal" />
</xs:simpleType>
```

In this case, you have a data type named `semList` that contains a list of decimal values. To apply this new data type to the `semGPA` element, you would reference the data type in the definition as follows:

```
<xs:element name="semGPA" type="semList" />
```

Notice that the type value does not have the `xs` namespace prefix because `semList` is not part of the XML Schema vocabulary; it is a named simple type created by the schema author.

## Deriving a Union Data Type

A **union data type** is based on the value and/or lexical spaces from two or more preexisting data types. Each base data type is known as a **member data type**. The syntax for deriving a union data type is

```
<xs:simpleType name="name">
  <xs:union memberTypes="type1 type2 type3 ..." />
</xs:simpleType>
```

where `type1`, `type2`, `type3`, etc., are the member types that constitute the union. XML Schema also allows unions to be created from nested simple types. The syntax is

```
<xs:simpleType name="name">
  <xs:union>
    <xs:simpleType>
      rules1
    </xs:simpleType>
    <xs:simpleType>
      rules2
    </xs:simpleType>
    ...
  </xs:union>
</xs:simpleType>
```

where `rules1`, `rules2`, etc., are rules for creating different user-derived data types. For example, when collecting data on semester GPA values, Sabrina might want to specify the type of semester GPA, such as `program`, `genEd`, or `all`. As a result of this variety, the `semGPA` element might look as follows:

```
<semGPA>3.81 program 3.92 all 3.3 genEd 3.2 all</semGPA>
```

To validate this element, which contains a mixture of numeric and descriptive measures, she could create the following derived data type:

```
<xs:simpleType name="semType">
  <xs:union memberTypes="xs:decimal xs:Name" />
</xs:simpleType>
```

Based on this simple type definition, a parser will accept any value as long as it is either a decimal value or a text string of the `Name` data type. Next, Sabrina would use this data type to derive a list type based on the following union data type:

```
<xs:simpleType name="semList">
  <xs:list itemType="semType" />
</xs:simpleType>
```

This list data type would allow the `semGPA` element to contain a list consisting of either decimal values or XML names.

## Deriving a Restricted Data Type

The final kind of derived data type is a restricted data type, in which a restriction is placed on the facets of a preexisting data type, such as an `integer` data type that is constrained to fall within a range of values. XML Schema provides 12 constraining facets that can be used to derive new data types; these facets are described in Figure 3-28.

Figure 3-28

Constraining facets

Facet	Description
<code>enumeration</code>	Constrains the data to a specified list of values
<code>length</code>	Specifies the length of the data in characters (for text strings) or items (for lists)
<code>maxLength</code>	Specifies the maximum length of the data in characters (for text strings) or items (for lists)
<code>minLength</code>	Specifies the minimum length of the data in characters (for text strings) or items (for lists)
<code>pattern</code>	Constrains the lexical space of the data to follow a specific character pattern
<code>whiteSpace</code>	Controls the use of blanks in the lexical space of the data; the <code>whiteSpace</code> facet has three values— <code>preserve</code> (preserve all white space), <code>replace</code> (replace all tabs, carriage returns, and line feed characters with blank spaces), and <code>collapse</code> (collapse all consecutive occurrences of white space to a single blank space, and remove any leading or trailing white space)
<code>maxExclusive</code>	Constrains the data to be less than a maximum value
<code>maxInclusive</code>	Constrains the data to be less than or equal to a maximum value
<code>minExclusive</code>	Constrains the data to be greater than a minimum value
<code>minInclusive</code>	Constrains the data to be greater than or equal to a minimum value
<code>fractionDigits</code>	Specifies the maximum number of decimal places to the right of the decimal point in the data value
<code>totalDigits</code>	Specifies the maximum number of digits in the data value

Constraining facets are applied to a base type using the structure

```
<xs:simpleType name="name">
  <xs:restriction base="type">
    <xs:facet1 value="value1" />
    <xs:facet2 value="value2" />
    ...
  </xs:restriction>
</xs:simpleType>
```

where *type* is the data type on which the restricted data type is based; *facet1*, *facet2*, etc., are constraining facets; and *value1*, *value2*, etc., are values for the constraining facets. In Sabrina's document, each student's total credits must be at least 1. You could create a restricted data type using the *minInclusive* facet to restrict the *credits* value to at least 1, as in the following code:

```
<xs:simpleType name="creditsType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1" />
  </xs:restriction>
</xs:simpleType>
```

When applied to the *credits* element, this data type would require each *credits* value to be an integer with a minimum value of 1.

### INSIGHT

#### Constraining Facets vs Form Validation

Like form validation in HTML, constraining facets in an XML schema let you place limits on the allowable values for data in your document. However, while you can use HTML validation to prompt users to correct information they've entered, an XML parser uses a schema to decide whether or not an entire document is valid, and rejects the whole document if it does not adhere to all the rules of the schema, including the constraining facets for specific elements. In short, HTML form validation is a tool to ensure valid collection of data from users, while constraining facets in an XML schema generally serve only as a check on data that has already been collected.

You'll add the *creditsType* data type to the schema and apply it to the *credits* element next.

## REFERENCE

### Deriving Customized and Patterned Data Types

- To derive a list data type, use

```
<xs:simpleType name="name">
    <xs:list itemType="type" />
</xs:simpleType>
```

where *name* is the name of the custom data type and *type* is the data type on which it is based.

- To derive a union data type, use

```
<xs:simpleType name="name">
    <xs:union memberTypes="type1 type2 type3 ..." />
</xs:simpleType>
```

where *type1*, *type2*, *type3*, etc., are the member types that constitute the union and upon which the custom data type is based. Alternatively, you can use the nested form

```
<xs:simpleType name="name">
    <xs:union>
        <xs:simpleType>
            rules1
        </xs:simpleType>
        <xs:simpleType>
            rules2
        </xs:simpleType>
        ...
    </xs:union>
</xs:simpleType>
```

where *rules1*, *rules2*, etc., are the rules that define the different data types in the union.

- To derive a data type by restricting the values of a preexisting data type, use

```
<xs:simpleType name="name">
    <xs:restriction base="type">
        <xs:facet1 value="value1" />
        <xs:facet2 value="value2" />
        ...
    </xs:restriction>
</xs:simpleType>
```

where *facet1*, *facet2*, etc., are constraining facets; and *value1*, *value2*, etc., are values for the constraining facets.

- To derive a data type based on a regular expression pattern, use

```
<xs:simpleType name="name">
    <xs:restriction base="type">
        <xs:pattern value="regex" />
    </xs:restriction>
</xs:simpleType>
```

where *name* is the name of the derived data type, *type* is a preexisting data type on which the derived type is based, and *regex* is a regular expression defining the pattern of characters in the data.

### To create the creditsType data type:

- 1. Return to the **students.xsd** file in your editor.
- 2. Below the declaration for the **gpa** element, insert the following code to constrain the **credits** element to at least 1:

```
<xs:simpleType name="creditsType">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="1" />
    </xs:restriction>
</xs:simpleType>
```

- 3. In the **xs:element** tag for the **credits** element, change the value of the **type** attribute from **xs:positiveInteger** to **creditsType**. You do not include the **xs** namespace prefix when referencing the data type because **creditsType** is not part of the XML Schema vocabulary. Figure 3-29 shows the revised code in the schema.

Be sure to enter the data type as **creditsType** without a prefix, and *not* as **xs:creditsType** because the data type is user-defined.

Figure 3-29

### The creditsType data type

credits values must follow the rules of the creditsType simple type

restricted data type with a constraining facet that limits values to integers greater than or equal to 1

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:attribute name="stuID" type="xs:ID" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="dateOfEnrollment" type="xs:date" />
    <xs:element name="credits" type="creditsType" />
    <xs:element name="comment" type="xs:string" />
    <xs:attribute name="degree" type="xs:string" />

    <xs:element name="gpa">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:decimal">
                    <xs:attribute ref="degree" use="required" />
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name="creditsType">
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="1" />
        </xs:restriction>
    </xs:simpleType>
```

- 4. Save your changes to the **students.xsd** file.

Facets can also be used to define lower and upper ranges for data. In Sabrina's data, GPA values range from 0 to 4, with 0 excluded and 4 included as possible values. You'll create a data type for this interval now using the **minExclusive** and **maxInclusive** facets. You'll set the value of the **minExclusive** facet to 0, and set the value of the **maxInclusive** facet to 4. You'll name the data type **gpaType** and apply it to the **gpa** element.

### To derive the gpaType data type in the students.xsd file:

- 1. Below the code for the creditsType simple type, insert the following code:

```
<xs:simpleType name="gpaType">
    <xs:restriction base="xs:decimal">
        <xs:minExclusive value="0" />
        <xs:maxInclusive value="4" />
    </xs:restriction>
</xs:simpleType>
```

This code specifies decimal values greater than 0 and less than or equal to 4.

- 2. Within the code that defines the gpa element, locate the opening xs:extension tag and then change the value of the base attribute from xs:decimal to gpaType. Do not include the xs namespace prefix when referencing the data type. See Figure 3-30.

**Figure 3-30**

### The gpaType data type

gpa values must follow the rules of the gpaType simple type

```
<xs:element name="gpa">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="gpaType">
                <xs:attribute ref="degree" use="required" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>

<xs:simpleType name="creditsType">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="1" />
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="gpaType">
    <xs:restriction base="xs:decimal">
        <xs:minExclusive value="0" />
        <xs:maxInclusive value="4" />
    </xs:restriction>
</xs:simpleType>
```

restricted data type with a constraining facet that limits values to integers greater than 0, and up to and including 4

- 3. Save your changes to the file.

Sabrina wants values of the degree attribute to be limited to either MP or WPA. When permitted content belongs to a set of specific values rather than a range, you can create a list of possible values using the enumeration element. The following simple type creates the restriction that Sabrina needs:

```
<xs:simpleType name="degreeType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="MP" />
        <xs:enumeration value="WPA" />
    </xs:restriction>
</xs:simpleType>
```

You'll create this enumerated data type now and apply it to the schema.

### To create the degreeType data type in the students.xsd file:

- 1. Below the code that defines the gpaType simple type, insert the following code:
- ```
<xs:simpleType name="degreeType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="MP" />
        <xs:enumeration value="WPA" />
    </xs:restriction>
</xs:simpleType>
```
- 2. Within the xs:attribute element for the degree attribute, change the value of the type attribute from xs:string to degreeType. Do not include the xs namespace prefix when referencing the data type. Figure 3-31 highlights the revised code.

Figure 3-31

### The degreeType data type

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:attribute name="stuID" type="xs:ID" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="dateOfEnrollment" type="xs:date" />
    <xs:element name="credits" type="creditsType" />
    <xs:element name="comment" type="xs:string" />
    <xs:attribute name="degree" type="degreeType" />
    ...
    <xs:simpleType name="gpaType">
        <xs:restriction base="xs:decimal">
            <xs:minExclusive value="0" />
            <xs:maxInclusive value="4" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="degreeType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="MP" />
            <xs:enumeration value="WPA" />
        </xs:restriction>
    </xs:simpleType>
```

the degreeType data type limits values to "MP" and "WPA"

degree values must follow the rules of the degreeType simple type

- 3. Save your changes to the **students.xsd** file.

Before editing your schema further, you'll validate revisions to the schema and then revalidate it after introducing errors in the **students.xml** file and verifying that a parser flags the errors in validation.

### To validate revisions to the schema, introduce errors into the students.xml file, and then test the validation rules for the schema file:

- 1. Return to the **students.xml** file in Exchanger XML editor, and then, if necessary, click the **Viewer** button.
- 2. Validate your document and confirm that it validates successfully.

**Trouble?** If any validation errors are listed on the Errors tab, return to the students.xsd document and be sure your code matches the code shown in Figures 3-29 through 3-31 exactly. Make changes as needed, save your document, and then return to students.xml and revalidate the document until it passes.

- 3. Click the **Editor** button in the upper-right corner.
- 4. Locate the **student** element for Cynthia Bernstein, and then in the **gpa** element, change the value of the **degree** attribute to **MPA**. The data for the first **student** element in the document should match Figure 3-32.

Figure 3-32

**Invalid degree value**

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="students.xsd">
  <student stuID="SI890-041-02">
    <lastName>Berstein</lastName>
    <firstName>Cynthia</firstName>
    <dateOfEnrollment>2017-05-22</dateOfEnrollment>
    <credits>12</credits>
    <gpa degree="MPA">3.81</gpa>
  </student>
```

content changed to invalid  
value to introduce an error

- 5. Save your changes to the file.
- 6. On the Menu bar, click **XML**, click **Validate**, and then review the error descriptions displayed on the Errors tab. The document is rejected and the parser reports errors because the value of the attribute **degree** for the element **gpa** does not match the schema definitions.
- 7. In the **gpa** element for the first record, change the value of the **degree** attribute back to **MP**, and then save your changes to the file.
- 8. Revalidate the document and confirm that it passes.

## Deriving Data Types Using Regular Expressions

Sabrina has one final restriction to place on data values stored in her student records: Each student's student ID must be entered in the form SI####-##-##, where # is a digit from 0 to 9.

This rule involves the representation of the values, so you need to create a restriction based on the lexical space. One way of doing this is through a regular expression.

## Introducing Regular Expressions

A regular expression is a text string that defines a character pattern. Regular expressions can be created to define patterns for many types of data, including phone numbers, postal address codes, and e-mail addresses—and, in the case of Sabrina's document, student IDs. To apply a regular expression in a data type, you create the simple type

```
<xs:simpleType name="name">
  <xs:restriction base="type">
    <xs:pattern value="regex" />
  </xs:restriction>
</xs:simpleType>
```

where *regex* is a regular expression pattern.

The most basic pattern specifies the characters that must appear in valid data. For instance, the following regular expression requires that the value of the data type be the text string *ABC*:

```
<xs:pattern value="ABC" />
```

Any other combination of letters, including the use of lowercase letters, would be invalid.

Instead of a pattern involving specific characters, though, you usually want a more general pattern involving character types, which are representations of different kinds of characters. The general form of a character type is

*\char*

where *char* represents a specific character type. Character types can include digits, word characters (any uppercase or lowercase letter, any digit, or the underscore character (`_`)), boundaries around words, and white space characters. Figure 3-33 describes the code for representing each of these character types.

Figure 3-33 Regular expression character types

| Character Type | Description                                                                                    |
|----------------|------------------------------------------------------------------------------------------------|
| \d             | A digit from 0 to 9                                                                            |
| \D             | A non-digit character                                                                          |
| \w             | A word character (an upper- or lowercase letter, a digit, or an underscore ( <code>_</code> )) |
| \W             | A non-word character                                                                           |
| \b             | A boundary around a word (a text string of word characters)                                    |
| \B             | The absence of a boundary around a word                                                        |
| \s             | A white space character (a blank space, tab, new line, carriage return, or form feed)          |
| \S             | A non-white space character                                                                    |
| .              | Any character                                                                                  |

### TIP

In a regular expression, the opposite of a character type is indicated by a capital letter. So while `\d` represents a single digit, `\D` represents any character that is not a digit.

For example, the character type for a single digit is `\d`. To create a regular expression representing three digits, you would apply the following pattern:

```
<xs:pattern value="\d\d\d" />
```

Any text string that contains three digits would match this pattern. Thus, the text strings 012 and 921 would both match this pattern, but 1,020 and 54 would not.

For more general patterns, characters can also be grouped into lists called **character sets** that specify exactly what characters or ranges of characters are allowed in the pattern.

The syntax of a character set is

`[chars]`

where *chars* is the set of characters in the character set. For example, the pattern

`<xs:pattern value="[dog]" />`

matches any of the characters d, o, or g. Because characters can be sorted alphabetically or numerically, a character set can also be created for a range of characters using the general syntax

`[char1-charN]`

where *char1* is the first character in the range and *charN* is the last character in the range. To create a range of lowercase letters, you would use the following pattern:

`<xs:pattern value="[a-z]" />`

Any lowercase letter would be matched by this pattern. You can also match numeric ranges. The following pattern matches any digit from 1 to 5:

`<xs:pattern value="[1-5]" />`

Figure 3-34 lists many of the common character sets used in regular expressions.

Figure 3-34

### Common regular expression character sets

| Character Set                             | Description                                                               |
|-------------------------------------------|---------------------------------------------------------------------------|
| <code>[<i>chars</i>]</code>               | Match any character in the <i>chars</i> list                              |
| <code>[^<i>chars</i>]</code>              | Do not match any character in <i>chars</i>                                |
| <code>[<i>char1</i>-<i>charN</i>]</code>  | Match any character in the range <i>char1</i> through <i>charN</i>        |
| <code>[^<i>char1</i>-<i>charN</i>]</code> | Do not match any character in the range <i>char1</i> through <i>charN</i> |
| <code>[a-z]</code>                        | Match any lowercase letter                                                |
| <code>[A-Z]</code>                        | Match any uppercase letter                                                |
| <code>[a-zA-Z]</code>                     | Match any letter                                                          |
| <code>[0-9]</code>                        | Match any digit from 0 to 9                                               |
| <code>[0-9a-zA-Z]</code>                  | Match any digit or letter                                                 |

The regular expressions you've looked at so far have involved individual characters. To specify the number of occurrences for a particular character or group of characters, a **quantifier** can be appended to a character type or set. Figure 3-35 lists the different quantifiers used in regular expressions. Some of these quantifiers should be familiar from your work with DTDs.

Figure 3-35

### Regular expression quantifiers

| Quantifier              | Description                                                    |
|-------------------------|----------------------------------------------------------------|
| *                       | Repeat 0 or more times                                         |
| ?                       | Repeat 0 times or 1 time                                       |
| +                       | Repeat 1 or more times                                         |
| { <i>n</i> }            | Repeat exactly <i>n</i> times                                  |
| { <i>n</i> ,}           | Repeat at least <i>n</i> times                                 |
| { <i>n</i> , <i>m</i> } | Repeat at least <i>n</i> times but no more than <i>m</i> times |

**TIP**

On the web, you can find libraries of predefined regular expression patterns that can be applied to common text strings such as phone numbers, postal codes, credit card numbers, and Social Security numbers.

As you saw earlier, to specify a pattern of three consecutive digits, you can use the following regular expression:

```
\d\d\d
```

Alternatively, you can employ the quantifier {3} using the pattern

```
<xs:pattern value="\d{3}" />
```

which also defines a pattern of three digits. Likewise, to validate a string of uppercase characters of any length, you can use the \* quantifier, as follows:

```
<xs:pattern value="[A-Z]*" />
```

Similarly, the following pattern uses the quantifier {0,10} to allow for a text string of uppercase letters from 0 to 10 characters long:

```
<xs:pattern value="[A-Z]{0,10}" />
```

## Applying a Regular Expression

You have only scratched the surface of what regular expressions can do. The topic of regular expressions could fill an entire tutorial by itself. However, you have covered enough to be able to implement Sabrina's request that all student ID strings be in the format SI###-##-##, where # is a digit from 0 to 9. The pattern for this expression is

```
<xs:pattern value="SI\d{3}-\d{3}-\d{2}" />
```

where the character type \d represents a single digit, and the quantifiers {3} and {2} indicate that the digits must be in groups of three and two, respectively. Note that the characters SI at the start of the pattern must be matched literally, including capitalization; thus, a student ID value of Si448-996-22 would not match the pattern because the i isn't capitalized. You'll create a data type named **siType** now based on this pattern.

### To derive the **siType** data type based on a regular expression:

- 1. Return to the **students.xsd** file in your editor, and then directly below the declaration for the degree attribute, insert the following code:

```
<xs:simpleType name="siType">
  <xs:restriction base="xs:ID">
    <xs:pattern value="SI\d{3}-\d{3}-\d{2}" />
  </xs:restriction>
</xs:simpleType>
```

The base data type is **xs:ID**, indicating that this custom data type will be derived from unique ID values in the document.

- 2. Within the declaration of the **stuID** attribute, change the value of the **type** attribute from **xs:ID** to **siType**. Do not include the **xs** namespace prefix when referencing the data type.

Figure 3-36 shows the revised code for the schema file.

Figure 3-36

## Declaring and applying the siType data type

stuID values must follow the rules of the siType simple type

values must be unique IDs of the form SI###-###-##, where # is a digit

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:attribute name="stuID" type="siType" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="dateOfEnrollment" type="xs:date" />
  <xs:element name="credits" type="creditsType" />
  <xs:element name="comment" type="xs:string" />
  <xs:attribute name="degree" type="degreeType" />

  <xs:simpleType name="siType">
    <xs:restriction base="xs:ID">
      <xs:pattern value="SI\d{3}-\d{3}-\d{2}" />
    </xs:restriction>
  </xs:simpleType>
```

- 3. Save your changes to the **students.xsd** file.

Next, you'll validate Sabrina's document to ensure that all the student IDs match the pattern you defined.

## To validate the student IDs:

- 1. Return to the **students.xml** file in your XML editor, and then verify that you are in Editor view.
- 2. Within the record for Cynthia Berstein, change the capital letters **SI** in the **stuID** value to the lowercase letters **si** so it reads **si890-041-02**. This value doesn't meet the validation criteria for the **siType** data type because the first two letters are lowercase. See Figure 3-37.

Figure 3-37

## Invalid stuID value

first two letters of value changed to lowercase

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="students.xsd">
  <student stuID="si890-041-02">
    <lastName>Berstein</lastName>
    <firstName>Cynthia</firstName>
    <dateOfEnrollment>2017-05-22</dateOfEnrollment>
    <credits>12</credits>
    <gpa degree="MP">3.81</gpa>
  </student>
```

- 3. Save your changes to the file.
- 4. On the Menu bar, click **XML**, click **Validate**, and then examine the contents of the Errors tab. The document is rejected with errors found because the value of the attribute **stuID** for the element **student** no longer matches the schema definitions.
- 5. Change the **stuID** value for Cynthia Berstein back to **SI890-041-02**, and then save your changes to the file.
- 6. Revalidate the document and confirm that it passes the validation test.

**INSIGHT**

### Deriving Data Types and Inheritance

You can use XML Schema to create a library of new data types, and you can use each new data type you create as the base for creating yet another type. For example, you could start with a data type for integer values, use that to define a new data type for positive integers, use that to define a data type for positive integers between 1 and 100, and so forth. Note that unless you are creating a list or a union, every new type represents a restriction of one or more facets in a preexisting type.

In some cases, you might want to fix a facet so that any new data types based upon it cannot modify it. For example, if Sabrina defined the `credits` data type to have a maximum value of 130, she might want to prevent any data types based on `credits` from being able to change that maximum value. She could do so by applying the `fixed` attribute to the `maxInclusive` facet as follows:

```
<xs:maxInclusive value="130" fixed="true" />
```

As a result of this attribute, any data type based on the `credits` type would have its maximum value set at 130. Because fixing a facet makes a data type library less flexible, it should be used only in situations where changing the facet value would dramatically alter the original meaning of the data type.

You can also use XML Schema to prevent any new data types from being created from an existing data type. This is done using the `final` attribute

```
<xs:simpleType name="name" final="derivation">
  ...
</xs:simpleType>
```

where `name` is the name of the new data type and `derivation` indicates which methods of creating new data types are prohibited (`list`, `union`, `restriction`, or `all`). For example, the following code defines the `credits` data type and prohibits it from being used in deriving a new data type through a union or a list:

```
<xs:simpleType name="credits" final="union list">
  ...
</xs:simpleType>
```

When the `final` attribute is omitted or set to an empty text string, XML Schema allows for any kind of derivations of the original type.

Sabrina is pleased with the work you have done on creating a schema for the document describing the students enrolled in the information technology programs at Austin Technical College.

**REVIEW****Session 3.2 Quick Check**

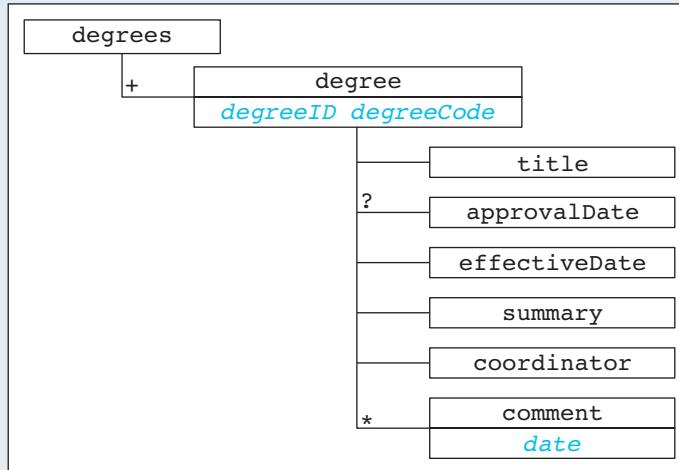
1. Enter a definition for an element named `Height` containing only decimal data.
2. Enter the attribute definition for the `productIDs` attribute containing a list of ID references.
3. Define a data type named `applicationDates` that contains a list of dates.
4. Define a data type named `status` that contains either a decimal or a text string.
5. Define a data type named `Party` that is limited to one of the following—  
Democrat, Republican, or Independent.
6. Define a data type named `Percentage` that is limited to decimal values falling between 0 and 1 (inclusive).
7. Define a data type named `socSecurity` that contains a text string matching the pattern `###-##-####`, where # is a digit from 0 to 9.

## Review Assignments

**Data Files needed for the Review Assignments:** `degreestxt.xml`, `degreestxt.xsd`

Each department at Austin Technical College must maintain information on the associate's degrees it offers; each associate's degree must be approved by a college authority. Sabrina has created an XML vocabulary containing information about the different degrees offered by the Information Technology department. She wants you to create a schema to validate the degree information. Figure 3-38 shows the structures of the degrees vocabulary that you'll create.

**Figure 3-38** The degrees vocabulary structure



A description of the elements and attributes used in the degrees vocabulary is shown in Figure 3-39.

**Figure 3-39** The degrees vocabulary

| Element or Attribute | Description                                                                |
|----------------------|----------------------------------------------------------------------------|
| degrees              | The root element                                                           |
| degree               | The collection of information about a degree                               |
| degreeID             | The ID number of the degree with the format IT##-##-##, where # is a digit |
| degreeCode           | The in-house code for the degree (MP, SP, or WPA)                          |
| title                | The title of the degree                                                    |
| approvalDate         | The date the degree was approved by the college authority                  |
| effectiveDate        | The date the degree curriculum becomes effective                           |
| summary              | The descriptive summary of the degree                                      |
| coordinator          | The currently assigned degree coordinator                                  |
| comment              | A comment regarding the degree                                             |
| date                 | The date of the comment                                                    |

Sabrina already created a document containing a list of degrees currently offered by the Information Technology department. She wants the document validated based on a schema you create.

Complete the following:

1. Using your XML editor, open the **degreestxt.xml** and **degreestxt.xsd** files from the **xml03** ▶ **review** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **degrees.xml** and **degrees.xsd**, respectively.
2. In the **degrees.xsd** file, add the root schema element to the document and declare the XML Schema namespace using the **xs** prefix, and then save your work.
3. In the **degrees.xml** file, attach the schema file **degrees.xsd** to this instance document, indicating that the schema and instance document do not belong to any namespace, and then save your work.
4. In the **degrees.xsd** file, create the following named simple types:
  - a. **idType**, based on the **ID** data type and restricted to the regular expression pattern **IT\d{2}-\d{3}-\d{3}**
  - b. **codeType**, based on the **string** data type and restricted to the following values—**MP**, **SP**, **WPA**
5. Declare the **degrees** element containing the child element **degree**.
6. Declare the **degree** element containing the following sequence of nested child elements—**title**, **approvalDate**, **effectiveDate**, **summary**, **coordinator**, and **comment**. Set the following properties for the nested elements:
  - a. All of the child elements should contain string data except the **approvalDate** and **effectiveDate** elements, which contain dates. The **degree** element should also support two required attributes—**degreeID** and **degreeCode**. The **degreeID** attribute contains **idType** data, while the **degreeCode** attribute contains **codeType** data.
  - b. The **degree** element must occur at least once, but its upper limit is unbounded. The **approvalDate** element is optional. The **comment** element is optional, and it may occur multiple times. All other elements are assumed to occur only once.
  - c. Each **comment** element requires a **date** attribute of the **date** data type.
7. Save your changes to the **degrees.xsd** file, and then validate the schema document. Correct any errors you find.
8. Validate the **degrees.xml** file against the schema document you created. Correct any validation errors you discover in the instance document.

**APPLY**

## Case Problem 1

**Data Files needed for this Case Problem: catalogtxt.xml, catalogtxt.xsd**

**The Our Lady of Bergen Historical Society** Sharon Strattan is an archivist at the Our Lady of Bergen Historical Society in Bergenfield, New Jersey. The historical society is exploring how to transfer its listings to XML format, and Sharon has begun by creating a sample document of the society's extensive collection of photos. As part of this process, she's asked for your help in developing the schema that will be used to validate the XML documents. She has created a sample document to work on. Eventually, your work will be used in a much larger system. The structure of the sample document is shown in Figure 3-40.

**Figure 3-40** The catalog vocabulary structure

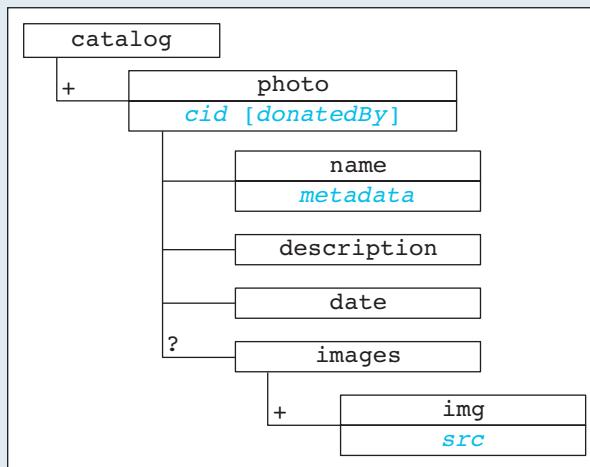


Figure 3-41 describes the elements and attributes in this sample document as well as the rules that govern the data that can be entered into a valid document.

**Figure 3-41** The catalog vocabulary

| Element or Attribute | Description                                                            |
|----------------------|------------------------------------------------------------------------|
| catalog              | The root element                                                       |
| photo                | The collection of information about a photo                            |
| cid                  | The ID number of the catalog with the format c####, where # is a digit |
| donatedBy            | The name of the donor                                                  |
| name                 | The name of the photo                                                  |
| metadata             | The metadata for the photo                                             |
| description          | The description of the photo                                           |
| date                 | The approximate date of the photo                                      |
| images               | The collection of img elements                                         |
| img                  | The element that references the image file                             |
| src                  | The source file containing the image; must end with .jpg               |

Your job will be to express this document structure and set of rules in terms of the XML Schema language, and then to validate Sharon's document based on the schema you create.

Complete the following:

1. Using your XML editor, open the **catalogtxt.xml** and **catalogtxt.xsd** files from the **xml03 ▶ case1** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **catalog.xml** and **catalog.xsd**, respectively.
2. Go to the **catalog.xsd** file in your text editor. Add the root schema element to the document and declare the XML Schema namespace using the **xs** prefix.
3. Attach the schema file **catalog.xsd** to the instance document, indicating that the schema and instance document do not belong to any namespace.

4. Create the following named simple types:
  - a. **cidType**, based on the **ID** data type and restricted to the regular expression pattern `c\d{4}`
  - b. **srcType**, based on the **string** data type and restricted to the regular expression pattern `[a-zA-Z0-9]+.jpg`
5. Declare the **catalog** element containing the child element **photo**. The **photo** element must occur at least once, but its upper limit is unbounded.
6. Declare the **photo** element containing the following sequence of nested child elements—**name**, **description**, **date**, and **images**. Set the following properties for the nested elements:
  - a. All of the child elements should contain string data. The **name** element should also support the **metadata** attribute.
  - b. The **cid** attribute is required. The **donatedBy** attribute is optional.
7. Declare the **img** element. It has no content and contains a required attribute, **src**.
8. Declare the following attributes and elements:
  - a. The attribute **metadata** must have the **string** data type.
  - b. The attribute **cid** must have the **cidType** data type.
  - c. The attribute **src** must have the **srcType** data type.
  - d. The attribute **donatedBy** must have the **string** data type.
  - e. The element **description** must have the **string** data type.
  - f. The element **date** must have the **string** data type.
9. Save your changes to the catalog.xsd file, and then validate the schema. Continue to correct any validation errors you discover until the schema validates.
10. Validate the catalog.xml file against the schema. Continue to correct any validation errors you discover until the instance document validates.

**APPLY**

## Case Problem 2

**Data Files needed for this Case Problem:** mdpbatxt.xml, mdpbatxt.xsd

**Midwest Developmental Pipe Band Association** Jacob St. John works as a coordinator for the Midwest Developmental Pipe Band Association (MDPBA) and is responsible for coordinating competitions for the MDPBA's many developmental pipe bands in the Midwest. Part of Jacob's job is to maintain a document that lists competition entries for each pipe band. As part of this process, he's asked for your help with developing the schema that will be used to validate the XML documents. He has created a sample document to work on. The structure of the sample document is shown in Figure 3-42.

**Figure 3-42** The bands vocabulary structure

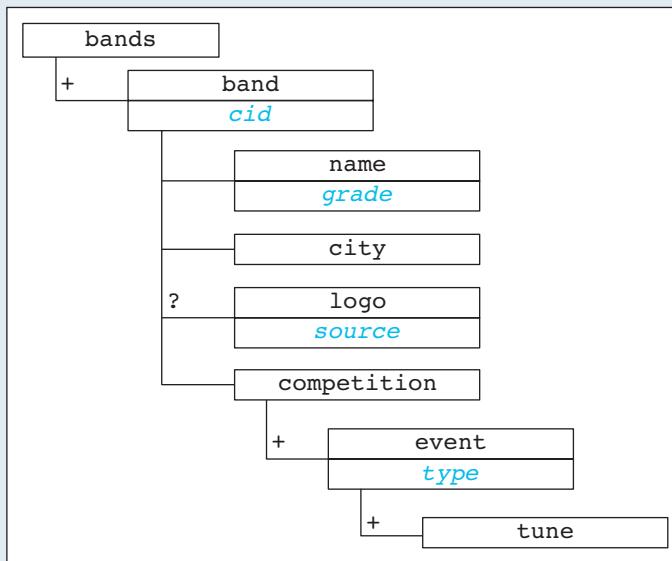


Figure 3-43 describes the elements and attributes in this sample document as well as the rules that govern the data that can be entered into a valid document.

**Figure 3-43** The bands vocabulary

| Element or Attribute | Description                                                                      |
|----------------------|----------------------------------------------------------------------------------|
| bands                | The root element                                                                 |
| band                 | The collection of information about a band                                       |
| cid                  | The ID number of the band with the format c####, where # is a digit              |
| name                 | The name of the band                                                             |
| grade                | The grade level at which the band is competing (juvenile, novice, 1, 2, 3, or 4) |
| city                 | The home city of the band                                                        |
| logo                 | The logo image                                                                   |
| source               | The source file containing the logo image                                        |
| competition          | The collection of information about a competition                                |
| event                | The collection of information about an event                                     |
| type                 | The type of event entered (MSR or Medley)                                        |
| tune                 | The name of a tune                                                               |

Your job will be to express this document structure and set of rules in terms of the XML Schema language, and then to validate Jacob's document based on the schema you create.

Complete the following:

1. Using your XML editor, open the **mdpbatxt.xml** and **mdpbatxt.xsd** files from the **xml03 ▶ case2** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **mdpba.xml** and **mdpba.xsd**, respectively.
2. Go to the **mdpba.xsd** file in your text editor. Add the root schema element to the document and declare the XML Schema namespace using the **xs** prefix.
3. Attach the schema file **mdpba.xsd** to the instance document, indicating that the schema and instance document do not belong to any namespace.

4. Create the following named simple types:
  - a. **gradeType**, based on the **string** data type and limited to the enumerated values **novice**, **juvenile**, 1, 2, 3, and 4
  - b. **eType**, based on the **string** data type and limited to the enumerated values **MSR** and **Medley**
  - c. **cidType**, based on the **ID** data type and restricted to the regular expression pattern **c\d{4}**
  - d. **srcType**, based on the **string** data type and restricted to the regular expression pattern **[a-zA-Z0-9]+.png**
5. Declare the **bands** element containing the child element **band**. The **band** element must occur at least once, but its upper limit is unbounded.
6. Declare the **band** element containing the following sequence of nested child elements—**name**, **city**, **logo**, and **competition**. Set the following properties for the nested elements:
  - a. All of the child elements should contain string data. The **name** element should also support the required **grade** attribute.
  - b. The **logo** element is optional.
  - c. The **band** element must contain the **cid** attribute.
7. Declare the **logo** element, which has no content and contains the required attribute **source**.
8. Declare the **competition** element containing the child element **event**. The **event** element must occur at least once, but its upper limit is unbounded.
9. Declare the **event** element containing the child element **tune**. The **tune** element must occur at least once, but its upper limit is unbounded. The **event** element is required.
10. Declare the following attributes and elements:
  - a. The attribute **grade**, which uses the **gradeType** data type
  - b. The attribute **type**, which uses the **eType** data type
  - c. The attribute **cid**, which uses the **cidType** data type
  - d. The attribute **source**, which uses the **srcType** data type
  - e. The element **city**, which uses the **string** data type
  - f. The element **tune**, which uses the **string** data type
11. Save your changes to the **mdpba.xsd** file, and then validate the schema. Continue to correct any validation errors you discover until the schema validates.
12. Validate the **mdpba.xml** file against the schema. Continue to correct any validation errors you discover until the instance document validates.

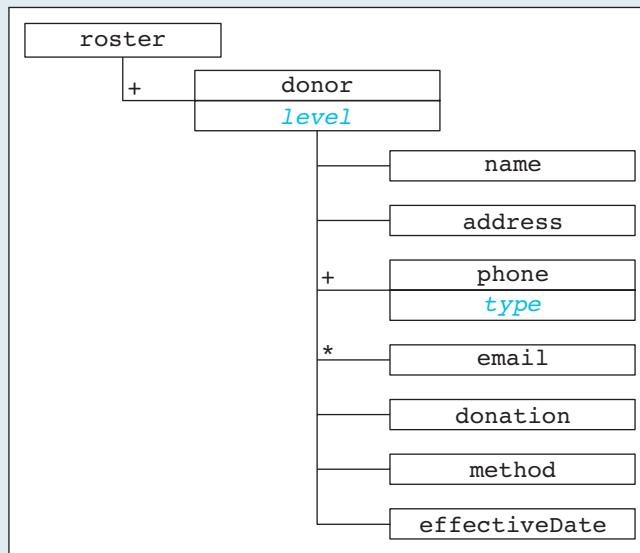
CREATE

### Case Problem 3

**Data Files needed for this Case Problem:** **rostertxt.xml**, **rostertxt.xsd**

**The Save Exotic Animals Trust** Sienna Woo is the donor coordinator for the Save Exotic Animals Trust (SEA Trust), a charitable organization located in central Florida. One of her responsibilities is to maintain a membership list of people in the community who have donated to SEA Trust. A donor can belong to one of four categories—friendship, patron, sponsor, or founder. A donor's phone number can be classified in one of three categories—home, cell, or work. Each donor's preferred method of contact can be one of three methods—Phone, Personal, or Mail. Sienna has asked for your help with developing a schema to validate the sample XML document that she created. The structure of the sample document is shown in Figure 3-44.

**Figure 3-44** The roster vocabulary structure



Your job will be to express this document structure in terms of the XML Schema language, and then to validate Sienna's document against the schema you create.

Complete the following:

1. Using your text editor, open the **rostertxt.xml** and **rostertxt.xsd** files from the **xml03 ▶ case3** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **roster.xml** and **roster.xsd**, respectively.
2. Go to the **roster.xsd** file in your text editor. Add the root schema element to the document and declare the XML Schema namespace using the **xs** prefix.
3. Attach the schema file **roster.xsd** to the instance document, indicating that the schema and instance document do not belong to any namespace.
4. Create the following named simple types:
  - a. **pType**, based on the **string** data type and limited to the enumerated values **home**, **cell**, and **work**
  - b. **methodType**, based on the **string** data type and limited to the enumerated values **Phone**, **Personal**, and **Mail**
  - c. **levelType**, based on the **string** data type and limited to the enumerated values **founder**, **sponsor**, **patron**, and **friendship**
  - d. **phoneType**, based on the **string** data type consisting of 14 characters—the first character should be a left opening parenthesis, followed by three digits from 0 to 9, followed by a right closing parenthesis, followed by a space, followed by three digits from 0 to 9, followed by a hyphen, and then four digits from 0 to 9 (*Hint: Opening and closing parentheses are special characters in creating regular expressions. To include one of these characters in your expression, enter a backslash before it.*)
5. Declare the **roster** element containing the child element listed in the vocabulary structure. The child element must occur at least once but its upper limit is unbounded.
  - a. Declare the **donor** element containing the sequence of nested child elements shown in the vocabulary structure. Set properties for the minimum and/or maximum occurrences of the **donor**, **phone**, and **email** elements as illustrated in the vocabulary structure. Specify that the **donor** element must contain the required **level** attribute.
  - b. Declare the **phone** element, containing the required attribute **type**.

6. Declare the following attributes and elements:
  - a. The attribute `type`, which uses the `pType` data type
  - b. The attribute `level`, which uses the `levelType` data type
  - c. The element `name`, which uses the `string` data type
  - d. The element `address`, which uses the `string` data type
  - e. The element `email`, which uses the `string` data type
  - f. The element `donation`, which uses the `decimal` data type
  - g. The element `effectiveDate`, which uses the `date` data type
  - h. The element `method`, which uses the `methodType` data type
7. Save your changes to the `roster.xsd` file, and then validate the schema. Continue to correct any validation errors you discover until the schema validates.
8. Validate the `roster.xml` file against the schema. In response to any validation errors, correct relevant values in the `roster.xml` document to match the schema rules. Continue to correct any validation errors you discover until the instance document validates.

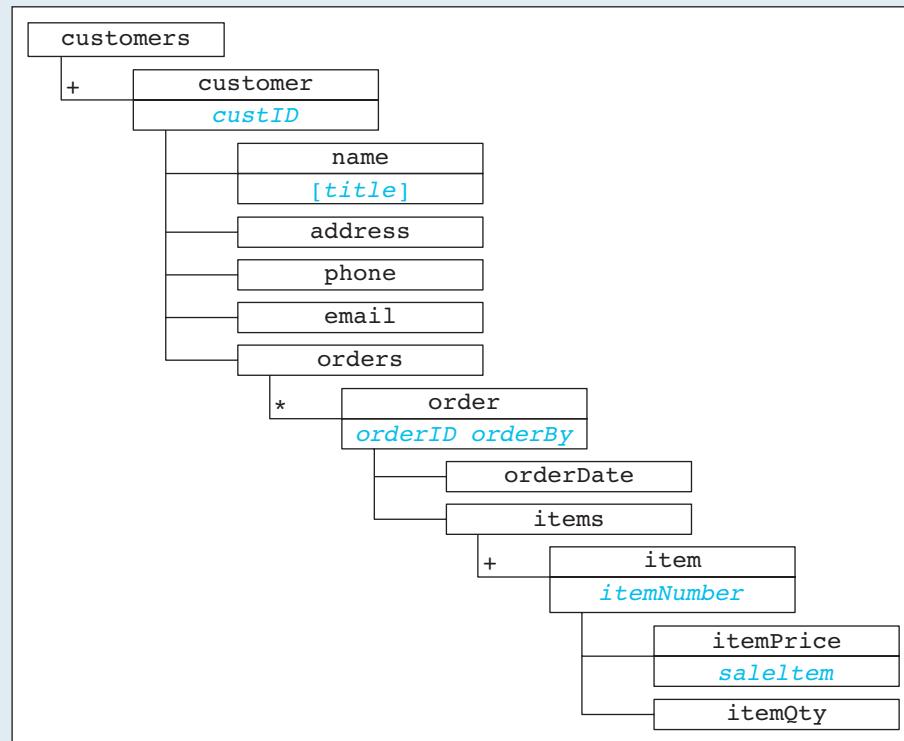
**CHALLENGE****Case Problem 4**

**Data Files needed for this Case Problem:** `orderstxt.xml`, `orderstxt.xsd`

**Map Finds For You** Benjamin Mapps is working on an XML document to hold data regarding customers who have placed orders with his store, Map Finds For You. Figure 3-45 shows the structure of the vocabulary employed by the document.

Figure 3-45

The customers vocabulary structure



A description of the elements and attributes used for customer data is shown in Figure 3-46.

**Figure 3-46** The customers vocabulary

| Element or Attribute    | Description                                                                               |
|-------------------------|-------------------------------------------------------------------------------------------|
| <code>customers</code>  | The root element                                                                          |
| <code>customer</code>   | The collection of information about a customer                                            |
| <code>custID</code>     | The ID number of the customer with the format <code>cust##</code> , where # is a digit    |
| <code>name</code>       | The name of the customer                                                                  |
| <code>title</code>      | The title for the customer ( <code>Mr.</code> , <code>Ms.</code> , or <code>Mrs.</code> ) |
| <code>address</code>    | The address for the customer                                                              |
| <code>phone</code>      | The phone number for the customer                                                         |
| <code>email</code>      | The email address for the customer                                                        |
| <code>orders</code>     | The collection of information on orders                                                   |
| <code>order</code>      | The collection of information on an individual order                                      |
| <code>orderID</code>    | The ID number of the order with the format <code>or##</code> , where # is a digit         |
| <code>orderBy</code>    | The ID number of the customer who placed the order                                        |
| <code>orderDate</code>  | The date the order was placed                                                             |
| <code>items</code>      | The collection of information on items                                                    |
| <code>item</code>       | The collection of information on a specific item                                          |
| <code>itemNumber</code> | The item number of the item ordered                                                       |
| <code>itemPrice</code>  | The price paid for the item                                                               |
| <code>saleItem</code>   | Whether the itemPrice was a sale price ( <code>Y</code> or <code>N</code> )               |
| <code>itemQty</code>    | The quantity of the item ordered                                                          |

Benjamin needs your help with creating a schema that will validate the data he has already entered and will enter in the future.

Complete the following:

1. Using your text editor, open the `orderstxt.xml` and `orderstxt.xsd` files from the `xml03 ▶ case4` folder, enter `your name` and `today's date` in the comment section of each file, and then save the files as `orders.xml` and `orders.xsd`, respectively.
2. Go to the `orders.xsd` file in your text editor and insert the root schema element. Declare the XML Schema namespace with `xs` as the namespace prefix.

 **EXPLORE** 3. Create the following simple data types:

- a. `idType`, based on the `ID` data type and consisting of the characters `cust` followed by three digits from 0 to 9
- b. `cidType`, based on the `ID` data type and consisting of the characters “`or`” followed by four digits from 0 to 9
- c. `titleType`, based on the `string` data type and limited to the enumerated values `Mr.`, `Ms.`, and `Mrs.`.

 **EXPLORE** d. `phoneType`, based on the `string` data type consisting of a left parenthesis followed by three digits from 0 to 9, followed by a right parenthesis, followed by a space, followed by three digits from 0 to 9, followed by a hyphen, and then four more digits 0 to 9 (*Hint:* Opening and closing parentheses are special characters in creating regular expressions. To include one of these characters in your expression, enter a backslash before it.)

- e. **qtyType**, based on the **integer** data type and allowing only numbers with a value of 1 or more
  - f. **saleType**, based on the **string** data type and limited to the values **y** and **n**
4. Declare the **customers** complex element type, and then nest the **customer** element within it. The **customer** element must occur at least once, but its upper limit is unbounded.

⊕ **EXPLORE** 5. Set the data types of the elements and attributes of the **customer** element as follows:

- a. The child elements must occur in the sequence **name**, **address**, **phone**, **email**, **orders**. The **name**, **address**, and **email** elements use the **string** data type. The **phone** element uses the **phoneType** data type. The **email** element can occur once or not at all.
- b. The **name** element is a complex type element and uses a data type of **string**. The **name** attribute also contains an optional **title** attribute.
- c. The **orders** element is a complex type element and contains at least one **order** element.
- d. Declare an attribute named **custID**. The **custID** attribute is required and contains **idType** data.

⊕ **EXPLORE** 6. Set the data types of the elements and attributes of the **order** element as follows:

- a. The child elements must occur in the sequence **orderDate**, **items**. The **orderDate** element uses the **date** data type.
- b. The **order** element also contains required **orderId** and **orderBy** attributes containing **cidType** and **IDREF** data types, respectively.
- c. The **items** element is a complex type element and contains at least one **item** element.

⊕ **EXPLORE** 7. Set the data types of the elements and attributes of the **item** element as follows:

- a. The child elements must occur in the sequence **itemPrice**, **itemQty**. The **itemPrice** and **itemQty** elements use the **decimal** and **qtyType** data types, respectively.
  - b. The **item** element also contains a required **itemNumber** attribute that uses the **string** data type.
  - c. The **itemPrice** element is a complex type element and contains an attribute, **saleItem**, which uses the **saleType** data type and has a default value of **n**.
8. Save your changes to the **orders.xsd** file, and then validate the schema. Continue to correct any validation errors you discover until the schema validates.
9. Validate the **orders.xml** file against the schema. In response to any validation errors, correct relevant values in the **orders.xml** document to match the schema rules. Continue to correct any validation errors you discover until the instance document validates.



**OBJECTIVES****Session 4.1**

- Explore the Flat Catalog schema design
- Explore the Russian Doll schema design
- Explore the Venetian Blind schema design

**Session 4.2**

- Attach a schema to a namespace
- Apply a namespace to an instance document
- Import one schema file into another
- Reference objects from other schemas

**Session 4.3**

- Declare a default namespace in a style sheet
- Specify qualified elements by default in a schema
- Integrate a schema and a style sheet with an instance document

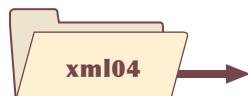
# Working with Advanced Schemas

*Creating Advanced Schemas for Higher Ed Test Prep*

## Case | **Higher Ed Test Prep**

Gabby Phelps is an exam study coordinator at Higher Ed Test Prep, an Internet-based company that prepares students for academic exams such as the PSAT, ACT, SAT, and GRE. Gabby wants to use XML to create structured documents containing information about the different exams that she oversees and about the students who are studying for those exams.

Accuracy is important for Higher Ed, so Gabby is using a schema to make sure that the data she enters is error free. Although she has already created a basic schema document, she would like to explore some different schema designs. Gabby also needs to create compound documents from the various XML vocabularies she's created.

**STARTING DATA FILES**

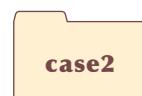
coursetxt.css  
coursetxt.xsd  
psattxt.xml  
studentsfctxt.xsd  
studentsrdtxt.xsd  
studentstxt.css  
studentstxt.xml  
studentsvctxt.xsd



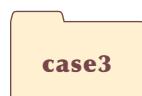
coursetxt.css  
coursetxt.xsd  
psattxt.xml  
sessionstxt.css  
sessionstxt.xml  
sessionstxt.xsd



sitemapPS.xml  
sitemapVS.xml  
sitemapWFS.xml  
sitestxt.xml  
sitestxt.xsd



menutxt.css  
menutxt.xml  
recipetxt.css  
recipetxt.xml



atclectxt.xml  
ituneselem.txt



carstxt.css  
carstxt.xml  
teamstxt.css  
teamstxt.xml

# Session 4.1 Visual Overview:

In a **Flat Catalog design**—sometimes referred to as a *Salami Slice design*—all element and attribute definitions have global scope.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="students">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="student" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="lastName" />
        <xs:element ref="firstName" />
        <xs:element ref="examDate" />
        <xs:element ref="pretest" />
        <xs:element ref="score" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute ref="stuID" use="required" />
      <xs:attribute ref="courseID" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="lastName" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="examDate" type="xs:date" />
  <xs:element name="score" type="pretestType" />
  <xs:element name="pretest">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="pretestType">
          <xs:attribute ref="level" use="required" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="pretestType">
    <xs:restriction base="xs:decimal">
      <xs:minExclusive value="0" />
      <xs:maxExclusive value="80" />
    </xs:restriction>
  </xs:simpleType>

  <xs:attribute name="stuID">
    <xs:simpleType>
      <xs:restriction base="xs:ID">
        <xs:pattern value="I\d{4}-\d{3}" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>

  <xs:attribute name="courseID">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]{4}-\d{3}-\d" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>

  <xs:attribute name="level">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="L" />
        <xs:enumeration value="M" />
        <xs:enumeration value="H" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:schema>
```

An object with **global scope** is a direct child of the root schema element and can be referenced throughout the schema document. Each code block shaded purple in this Visual Overview has global scope.

A **Russian Doll design** has only one global element with everything else nested inside of it, much like Russian dolls nest one inside another.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="students">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="student" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="lastName" type="xs:string" />
        <xs:element name="firstName" type="xs:string" />
        <xs:element name="examDate" type="xs:date" />
        <xs:element name="pretest">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="pretestType">
                <xs:attribute name="level" use="required" />
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="stuID" use="required" />
      <xs:attribute name="courseID" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="pretestType">
    <xs:restriction base="xs:decimal">
      <xs:minExclusive value="0" />
      <xs:maxExclusive value="80" />
    </xs:restriction>
  </xs:simpleType>

  <xs:attribute name="stuID">
    <xs:simpleType>
      <xs:restriction base="xs:ID">
        <xs:pattern value="I\d{4}-\d{3}" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>

  <xs:attribute name="courseID">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]{4}-\d{3}-\d" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>

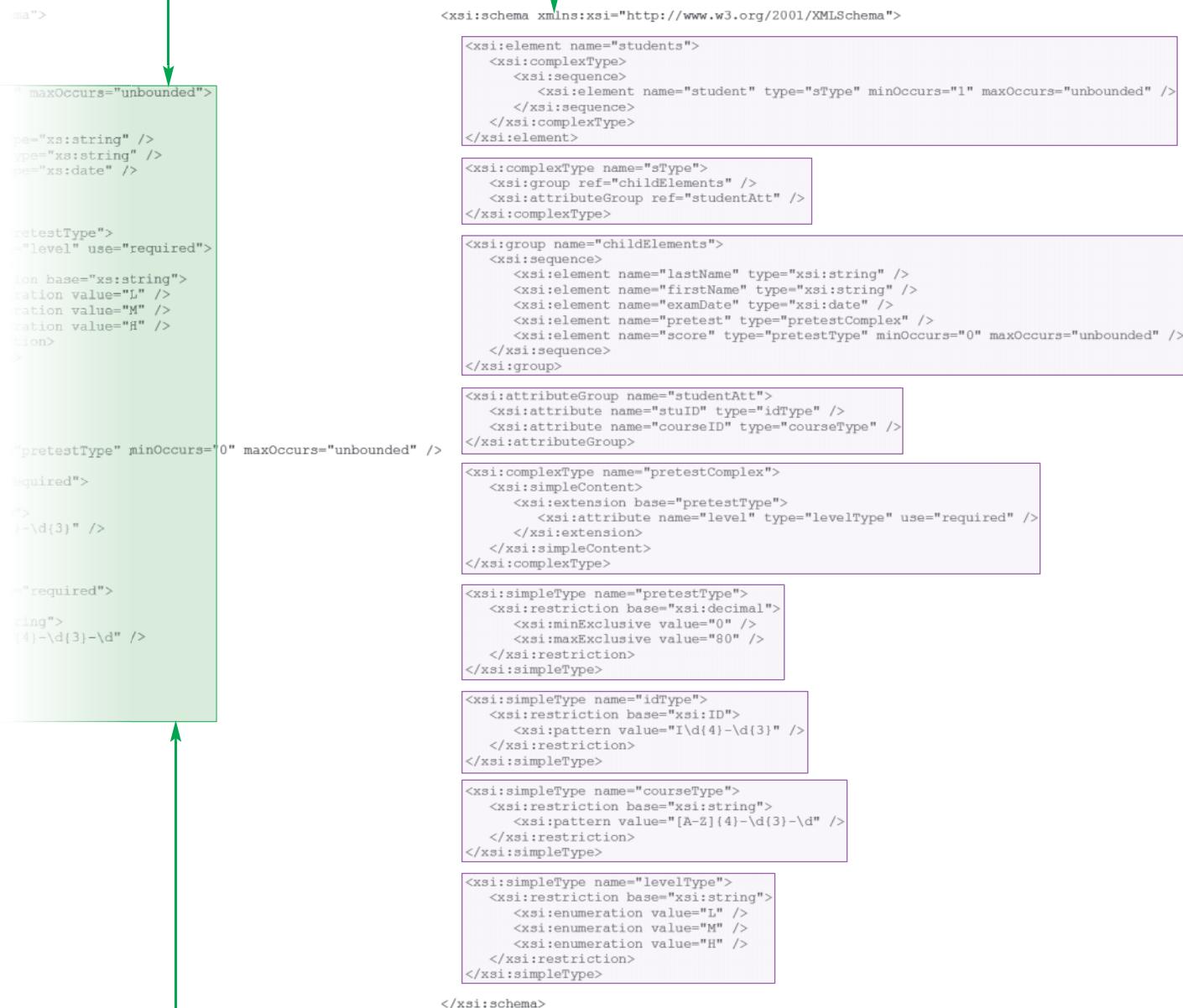
  <xs:attribute name="level">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="L" />
        <xs:enumeration value="M" />
        <xs:enumeration value="H" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:schema>
```

Russian Doll designs have very few if any global scope declarations. This declaration is the only one other than the root element with global scope in this Russian Doll design schema.

# Schema Design Comparisons

An object with **local scope** can only be referenced within the object in which it is defined. This code block, shaded green, is the only code with local scope in these schema designs.

A **Venetian Blind design** creates named types and references those types within a single global element.



A Russian Doll design is often compact, but the multiple levels of nested elements can be confusing and can make it more difficult to debug; it also means the nested element and attribute declarations cannot be reused elsewhere in the schema because they are made locally.

Globally defined complexTypes, simpleTypes, element groups, and attribute groups are used in a Venetian Blind design rather than globally defined elements and attributes. The only exception is the root element global definition.

## Designing a Schema

You and Gabby meet to discuss the needs of Higher Ed Test Prep. She has provided you with `students.xml`, which is a file that contains a list of students enrolled in the PSAT Mathematics Course. She has also created three different versions of a schema for the students vocabulary and would like your help with choosing the most appropriate schema design for the needs of Higher Ed Test Prep.

There are many different ways to design a schema. The building blocks of any schema are the XML elements that define the structure; these are known collectively as objects. You can create objects such as named complex types and then reuse them through the schema file, or you can nest one complex type inside of another. The way you design the layout of your schema file can impact how that schema is interpreted and applied to the instance document.

One important issue in schema design is determining the scope of the different objects declared within the schema. XML Schema recognizes two types of scope—global and local. Objects with global scope are direct children of the root schema element and can be referenced throughout the schema document. One advantage of creating objects with global scope is that you can reuse code several times in the same schema file without having to rewrite it. Objects with local scope can be referenced only within the object in which they are defined. It can be an advantage to keep all definitions confined to a local scope rather than referencing them throughout a long document—especially in a large and sprawling schema file—to avoid having to keep track of a large set of global objects. This distinction between global and local scope leads to three basic schema designs—Flat Catalog, Russian Doll, and Venetian Blind.

### Flat Catalog Design

In a Flat Catalog design—sometimes referred to as a Salami Slice design—all element and attribute definitions have global scope. Every element and attribute definition is a direct child of the root schema element and thus has been defined globally. The developer can then use references to the set of global objects to build the schema.

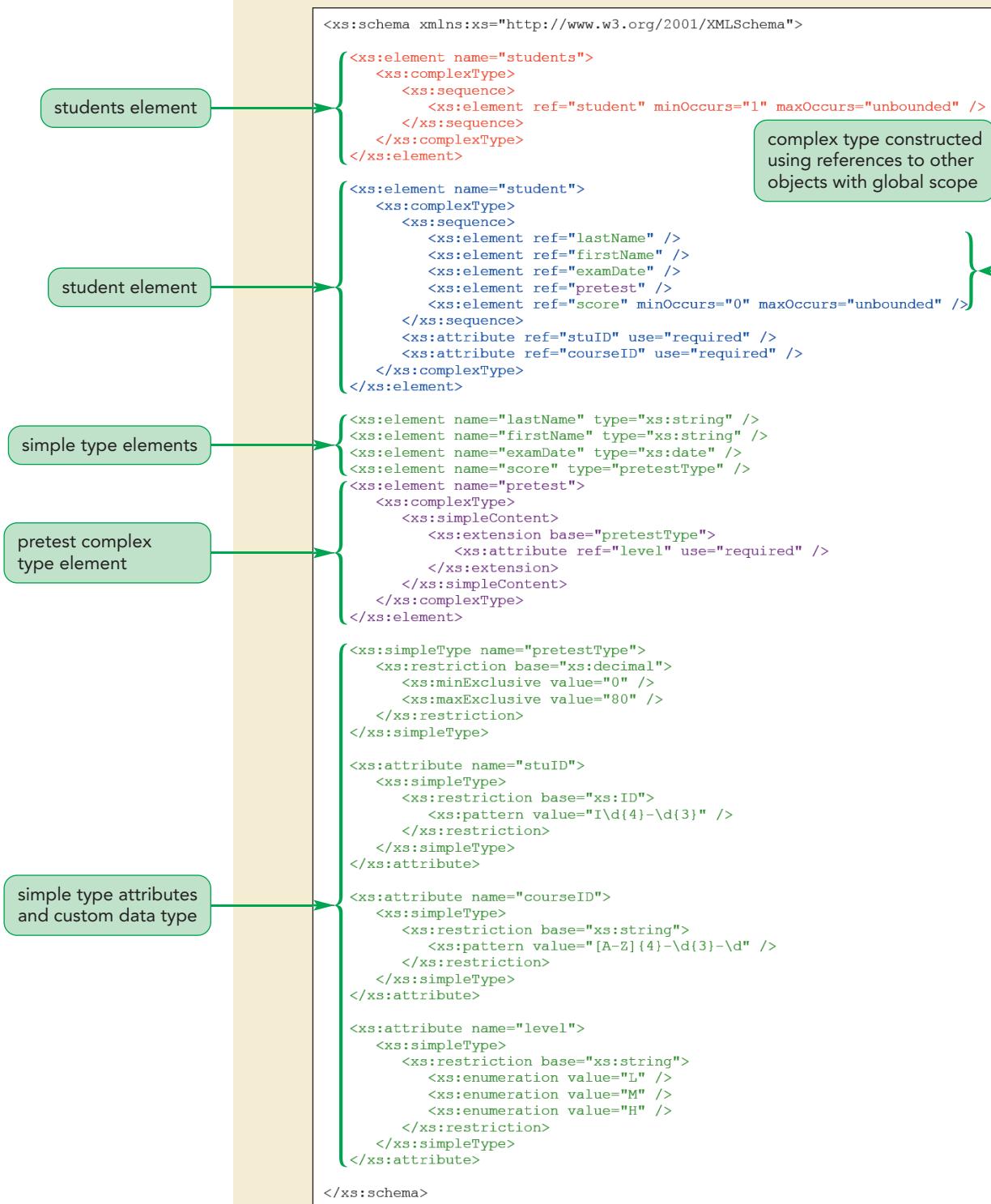
You'll open Gabby's `students.xml` file and the Flat Catalog version of the schema for the students vocabulary to explore what a schema that uses this design looks like.

#### To explore and apply the Flat Catalog version of the schema:

- 1. In your XML editor, open the **`studentstxt.xml`** and **`studentsfctxt.xsd`** files from the `xml04 ▶ tutorial` folder provided with your data files.
- 2. Within the comment section of each file, enter **`your name`** and **`today's date`**, and then save the files as **`students.xml`** and **`studentsfc.xsd`**, respectively. The `studentsfc.xsd` file is a Flat Catalog version of the schema for the students vocabulary.
- 3. Review the contents of the `studentsfc.xsd` document. As shown in Figure 4-1, all elements and attributes have global scope because they are all direct children of the root schema element.

Figure 4-1

## Schema for students vocabulary using Flat Catalog design



- 4. In the **students.xml** document, within the opening `<students>` tag, add the attribute `xsi:noNamespaceSchemaLocation="studentsfc.xsd"` on its own line, as shown in Figure 4-2. This attribute links the `students.xml` instance document to the `studentsfc.xsd` schema file.

Figure 4-2

**The students.xml document modified to use studentsfc.xsd schema**

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="studentsfc.xsd">
    <student stuID="I8900-041" courseID="PSAT-080-5">
```

name of the Flat Catalog schema document

- 5. Save your changes to the **students.xml** document and then validate it. The document validates successfully against the studentsfc.xsd schema.

## Russian Doll Design

A Russian Doll design has only one global element with everything else nested inside of it, much like Russian Matryoshka dolls nest one inside another. Russian Doll designs mimic the nesting structure of the elements in an instance document. The root element of the instance document becomes the top element declaration in the schema. All child elements within the root element are similarly nested in the schema. A Russian Doll design is much more compact than a Flat Catalog, but the multiple levels of nested elements can be confusing and can make it more difficult to debug. Also, the element and attribute declarations cannot be reused elsewhere in the schema because aside from the single root element, all object declarations are made locally.

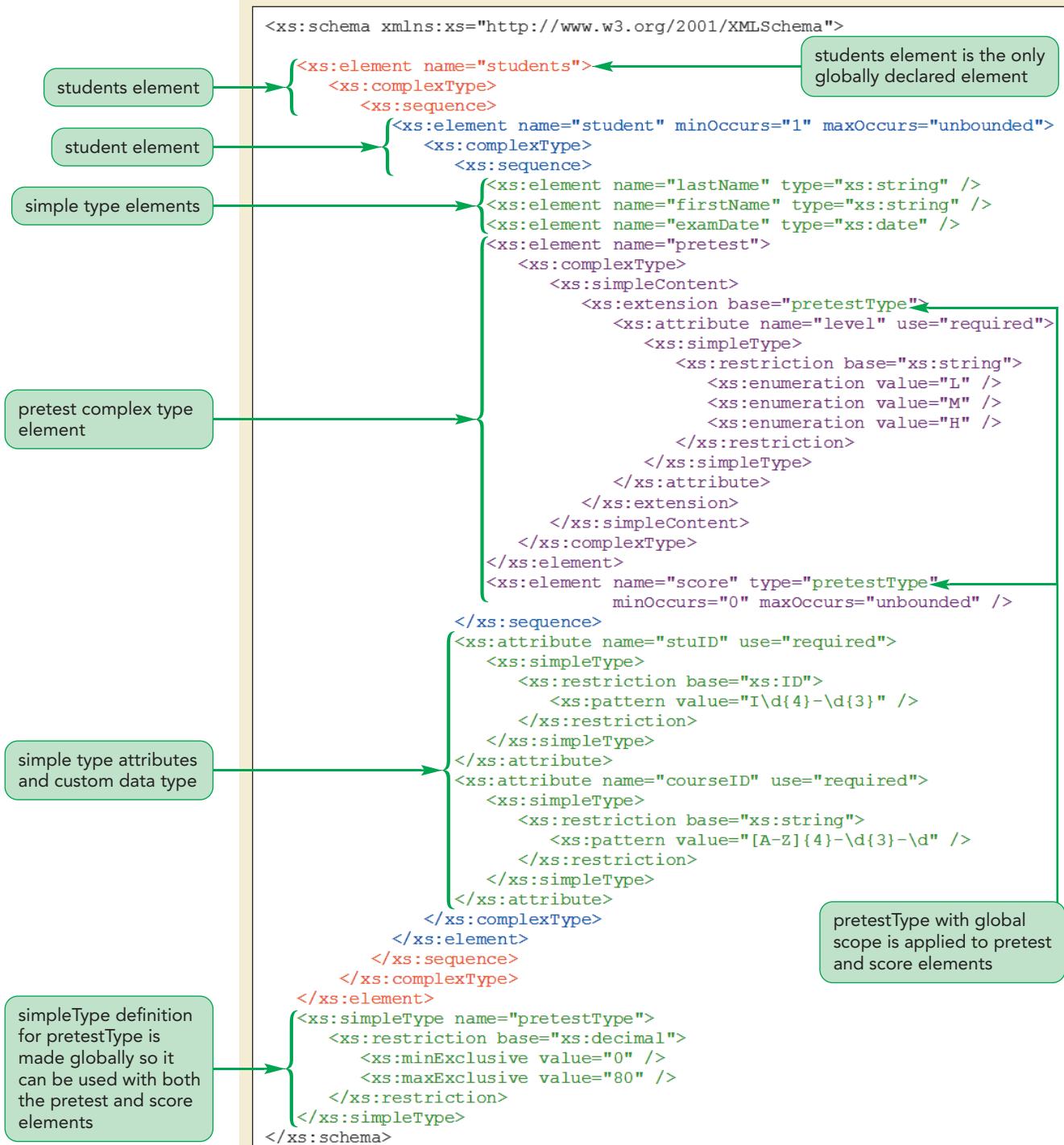
You'll explore the Russian Doll version of the schema for the students vocabulary now and apply it to the students.xml instance document.

**To explore and apply the Russian Doll version of the schema:**

- 1. In your XML editor, open the **studentsrddtxt.xsd** file from the xml04 ► tutorial folder provided with your data files.
- 2. Within the comment section, enter **your name** and **today's date**, and then save the file as **studentsrd.xsd**. The studentsrd.xsd file is a Russian Doll version of the schema for the students vocabulary.
- 3. Review the contents of the studentsrd.xsd document, as shown in Figure 4-3.

Figure 4-3

## Schema for students vocabulary using Russian Doll design



As Figure 4-3 shows, the only element declaration with global scope is for the `students` element; all other elements and attributes are declared locally, nested inside of the `students` element. Also notice that the definition for the `pretestType` simple type is global because it is defined outside the nested

part of the design. This allows the `pretestType` type to be reused in both the `pretest` and `score` elements. Because you can't set both a restriction and an extension to the same element definition, the only way to define the minimum and maximum values for the `pretest` element is by using a globally defined `simpleType` element. This is one area where strictly adhering to the Russian Doll design can be impractical.

- 4. Return to the **students.xml** document, and then change the `xsi:noNamespaceSchemaLocation` value from `studentsfc.xsd` to **`studentsrd.xsd`** as shown in Figure 4-4.

Figure 4-4

**The students.xml document modified to use the studentsrd.xsd schema**

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="studentsrd.xsd">
    <student stuID="I8900-041" courseID="PSAT-080-5">
```

name of the Russian Doll schema document

- 5. Save your changes to the **students.xml** document and then validate it. The document validates successfully against the `studentsrd.xsd` schema.

## Venetian Blind Design

A Venetian Blind design is similar to a Flat Catalog except that instead of declaring objects globally, it creates named types, named element groups, and named attribute groups and references those types within a single global element. A Venetian Blind design represents a compromise between Flat Catalogs and Russian Dolls. Although the various element and attribute groups and named types are declared globally (and can be reused throughout the schema), the declarations for the elements and attributes for the instance document are local and nested within element and attribute groups. The XML Schema `group` element is used to assign a name to a list of references to elements or attributes, and then the named group is referenced elsewhere using the `ref` attribute. Only the root element from the instance document—in this case, the `students` element—is defined globally.

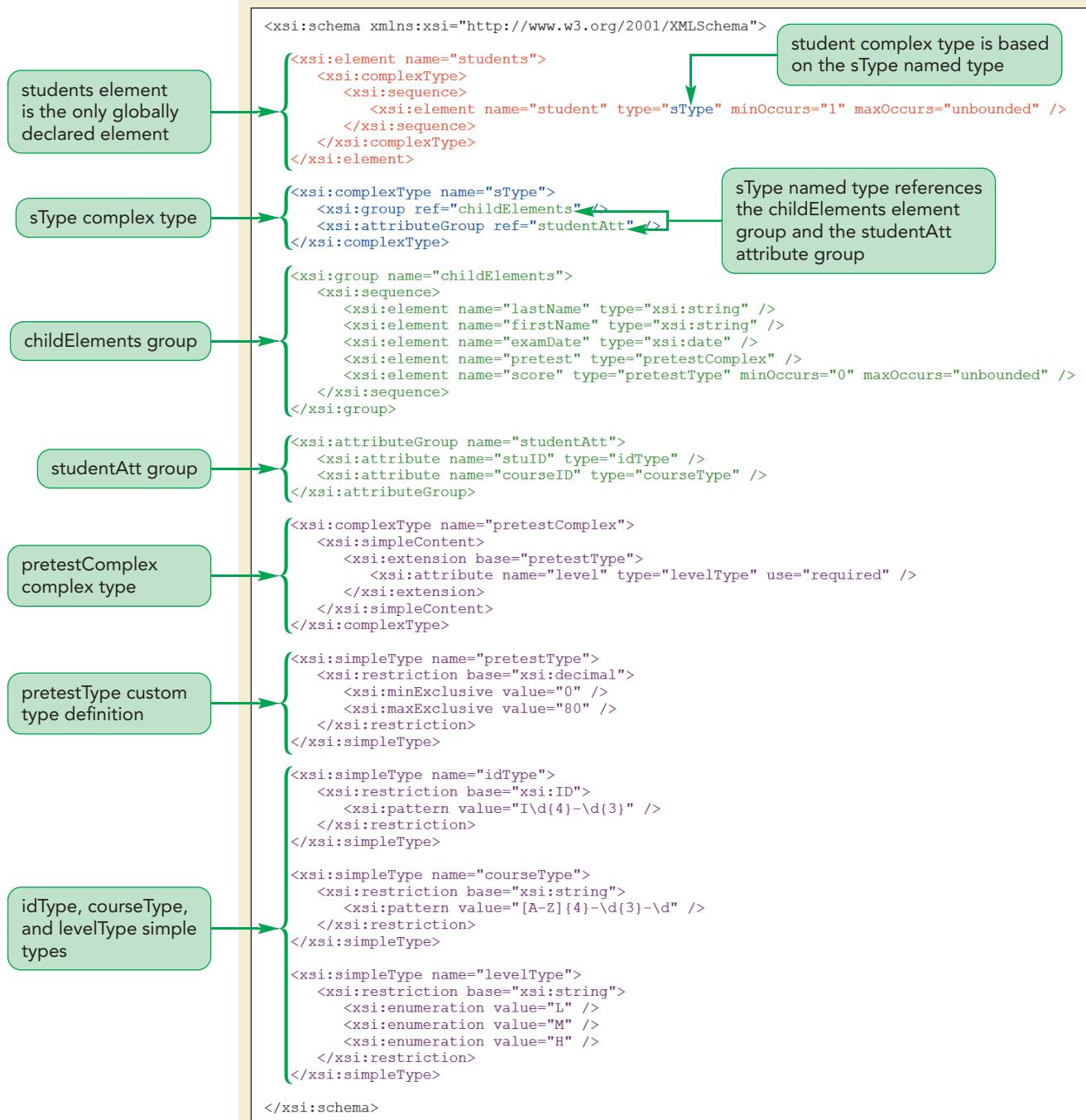
You'll explore the Venetian Blind version of the schema for the `students` vocabulary now and apply it to the `students.xml` instance document.

### To explore and apply the Venetian Blind version of the schema:

- 1. In your XML editor, open the **`studentsvbtxt.xsd`** file from the `xml04 ▶ tutorial` folder provided with your data files.
- 2. Within the comment section, enter **`your name`** and **`today's date`**, and then save the file as **`studentsvb.xsd`**. The `studentsvb.xsd` file is a Venetian Blind version of the schema for the `students` vocabulary.
- 3. Review the contents of the `studentsvb.xsd` document, noting the use of named types, element groups, and attribute groups, as shown in Figure 4-5.

Figure 4-5

## Schema for students vocabulary using Venetian Blind design



In this layout, the only globally declared element is the **students** element. All other elements and attributes are placed within element or attribute groups or, in the case of the **pretest** element's **level** attribute, within a named complex type.

4. Return to the **students.xml** document, and then change the **xsi:noNamespaceSchemaLocation** value from **studentsrd.xsd** to **studentsvb.xsd** as shown in Figure 4-6.

Figure 4-6

**The students.xml document modified to use the studentsvb.xsd schema**

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="studentsvb.xsd">
    <student stuID="I8900-041" courseID="PSAT-080-5">
```

name of the Venetian  
Blind schema document

- 5. Save your changes to the **students.xml** document and then validate it. The document validates successfully against the studentsvb.xsd schema.

**PROSKILLS*****Decision Making: Deciding Which Schema Design to Use***

Which schema layout you use depends on several factors. If a schema contains several lines of code that need to be repeated, you probably should use a Flat Catalog or Venetian Blind design. If you are interested in a compact schema that mirrors the structure of the instance document, you should use a Russian Doll design.

Figure 4-7 summarizes some of the differences among the three schema designs.

Figure 4-7

**Comparison of schema designs**

| Feature                       | Flat Catalog<br>(Salami Slice)                                                                        | Russian Doll                                                                                                    | Venetian Blind                                                                                                                        |
|-------------------------------|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| Global and local declarations | All declarations are global.                                                                          | The schema contains one single global element; all other declarations are local.                                | The schema contains one single global element; all other declarations are local.                                                      |
| Nesting of elements           | Element declarations are not nested.                                                                  | Element declarations are nested within a single global element.                                                 | Element declarations are nested within a single global element referencing named complex types, element groups, and attribute groups. |
| Reusability                   | Element declarations can be reused throughout the schema.                                             | Element declarations can only be used once.                                                                     | Named complex types, element groups, and attribute groups can be reused throughout the schema.                                        |
| Interaction with namespaces   | If a namespace is attached to the schema, all elements need to be qualified in the instance document. | If a namespace is attached to the schema, only the root element needs to be qualified in the instance document. | If a namespace is attached to the schema, only the root element needs to be qualified in the instance document.                       |

Rather than using the original Flat Catalog design for the schema file, Gabby wants you to continue to work with the Venetian Blind design. She likes the fact that the Venetian Blind layout maintains the flexibility of a Flat Catalog while providing a structure similar to the contents of her instance document. She has also heard that using a Venetian Blind design will make it easier to apply a namespace to the schema and instance document, which you'll explore in the next session.

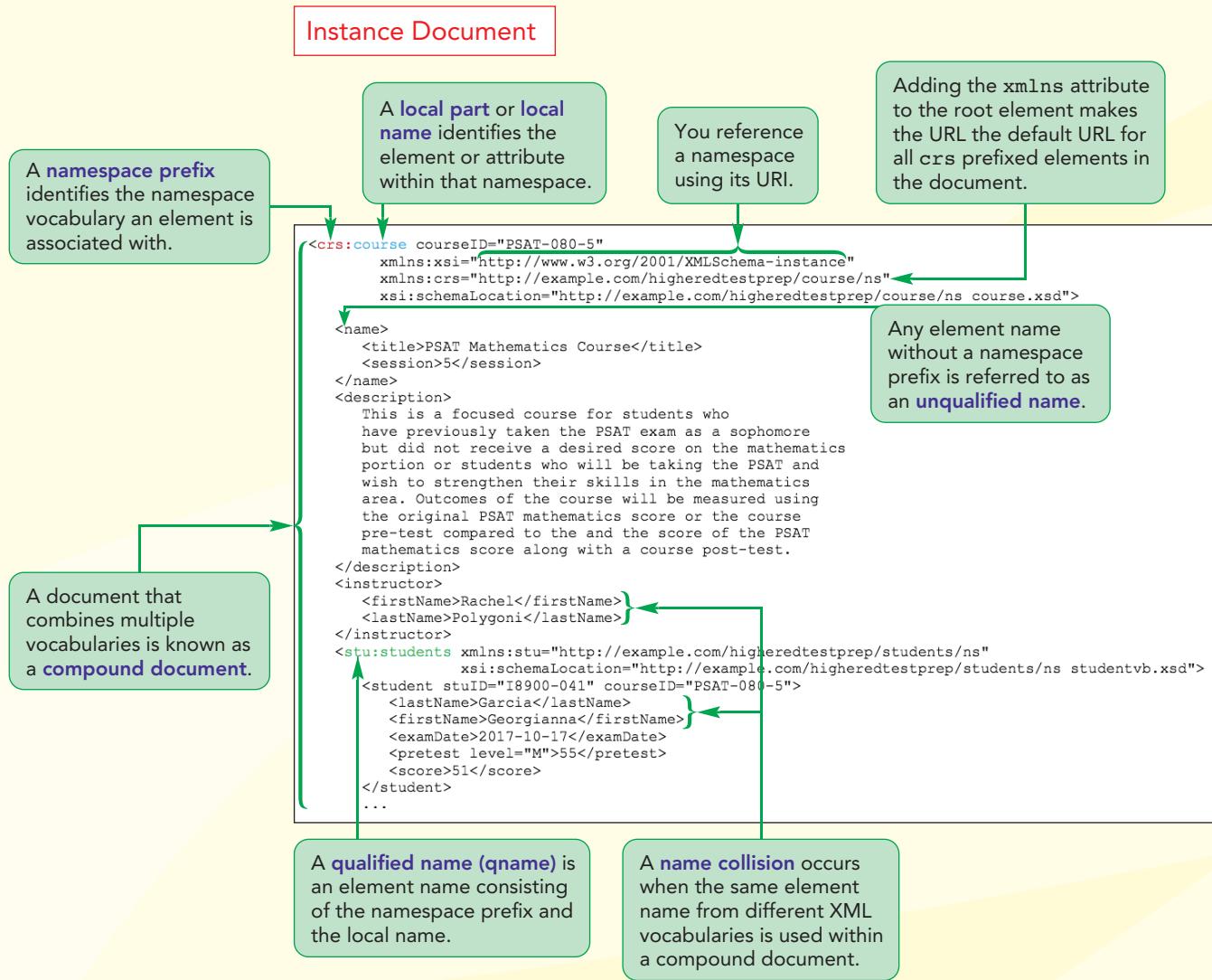
**INSIGHT*****The Garden of Eden Schema Design***

In addition to the Flat Catalog, Russian Doll, and Venetian Blind layouts, other standardized schema designs exist. One of these, known as the Garden of Eden design, combines the Flat Catalog approach of declaring all elements globally with the Venetian Blind practice of declaring type definitions globally. Although this results in a schema in which all parts are easily reusable, its main trade-off is that it requires more code than either of the designs on which it is based.

**REVIEW*****Session 4.1 Quick Check***

1. List and define the two types of scope for objects in a schema.
2. Name one advantage of each type of scope.
3. What is a Flat Catalog design and how does it differ from a Russian Doll design?
4. What is another name for the Flat Catalog design?
5. What is a Venetian Blind design, and how does it differ from the Flat Catalog and Russian Doll designs?

# Session 4.2 Visual Overview:



# A Compound Document

The **import** element combines schemas when the schemas come from different namespaces.

## Main Schema Document

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/course/ns"
    targetNamespace="http://example.com/higheredtestprep/course/ns"
    xmlns:stu="http://example.com/higheredtestprep/students/ns">

    <xss:import namespace="http://example.com/higheredtestprep/students/ns"
        schemaLocation="studentsvb.xsd" />

    <xs:element name="course">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="title" type="xs:string" />
                            <xs:element name="session" type="xs:string" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="description" type="xs:string" />
                <xs:element name="instructor">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="firstName" type="xs:string" />
                            <xs:element name="lastName" type="xs:string" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element ref="stu:students" />
            </xs:sequence>
            <xs:attribute name="courseID" type="xs:ID" />
        </xs:complexType>
    </xs:element>

</xs:schema>

```

The **ref** attribute is used to reference an object from an imported schema.

## Imported Schema Document

```

<xsi:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/students/ns"
    targetNamespace="http://example.com/higheredtestprep/students/ns">

    <xsi:element name="students">
        <xsi:complexType>
            <xsi:sequence>
                <xsi:element name="student" type="sType" minOccurs="1" maxOccurs="unbounded" />
            </xsi:sequence>
        </xsi:complexType>
    </xsi:element>

    <xsi:complexType name="sType">
        <xsi:group ref="childElements" />
        <xsi:attributeGroup ref="studentAtt" />
    </xsi:complexType>

    <xsi:group name="childElements">
        <xsi:sequence>
            <xsi:element name="lastName" type="xsi:string" />
            <xsi:element name="firstName" type="xsi:string" />
            <xsi:element name="examDate" type="xsi:date" />
            <xsi:element name="pretest" type="pretestComplex" />
            <xsi:element name="score" type="pretestType" minOccurs="0" maxOccurs="unbounded" />
        </xsi:sequence>
    </xsi:group>

    ...

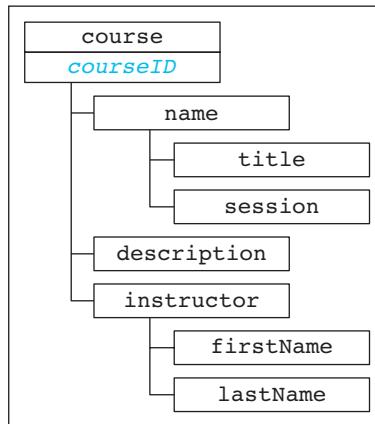
```

## Combining XML Vocabularies

Gabby has been working on a second XML vocabulary—one that documents the features of the different exam courses run by Higher Ed Test Prep. The structure of the course vocabulary is shown in Figure 4-8.

Figure 4-8

Structure of the course vocabulary



So far, the course vocabulary contains only basic information about each exam course: It records the course ID, name, title, session, and description of the course, and the first and last names of the course's instructor. As Gabby develops additional XML documents using this vocabulary, she will add more elements and attributes to it. Gabby already has created a schema file and an instance document based on this vocabulary. You will open both of those now.

### To view the schema file and instance document for the course vocabulary:

- 1. Use your XML editor to open the **coursetxt.xsd** file from the **xml04 ▶ tutorial** folder, enter **your name** and **today's date** in the comment section, and then save the file as **course.xsd**.
- 2. Review the contents and structure of the course schema shown in Figure 4-9. This schema uses a Russian Doll design, with **course** being the only element declared globally in the file.

Figure 4-9

## Schema for the course vocabulary

Russian Doll design

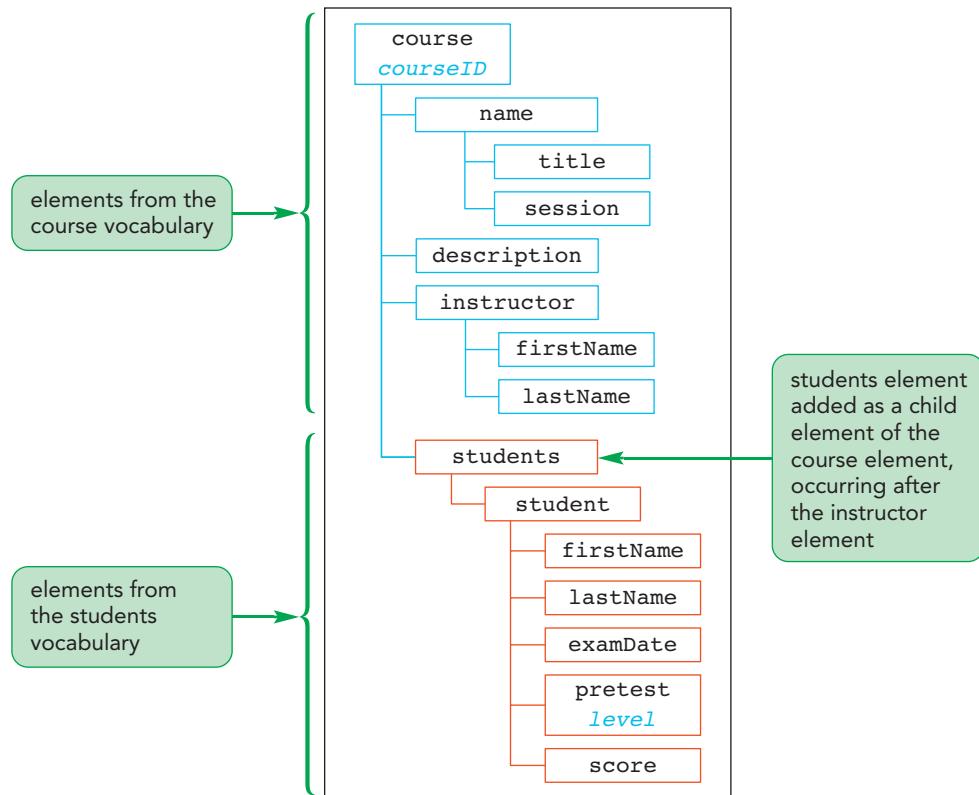
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="course">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string" />
              <xs:element name="session" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="description" type="xs:string" />
        <xs:element name="instructor">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstName" type="xs:string" />
              <xs:element name="lastName" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="courseID" type="xs:ID" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

course is the only element with global scope

- 3. Use your XML editor to open the **psattxt.xml** file from the **xml04 ▶ tutorial** folder, enter **your name** and **today's date** in the comment section, and then save the file as **psat.xml**. The psat.xml file contains basic information on a course that helps students improve their PSAT mathematics scores.

Gabby wants to combine the information about the PSAT Mathematics Course and the list of students enrolled in that course in a single compound document. A **compound document** is a document that combines elements from multiple vocabularies. Figure 4-10 shows a schematic diagram of the document Gabby wants to create, involving elements and attributes from both the course and students vocabularies.

**Figure 4-10 Structure of the compound document**



## Creating a Compound Document

Gabby wants you to work with her XML files to create a sample compound document that she can use as a model for future projects. You'll start by combining the elements from the students and course vocabularies, storing the result in a new file named psatstudents.xml. You'll create this compound document now.

### To create the psatstudents.xml compound document:

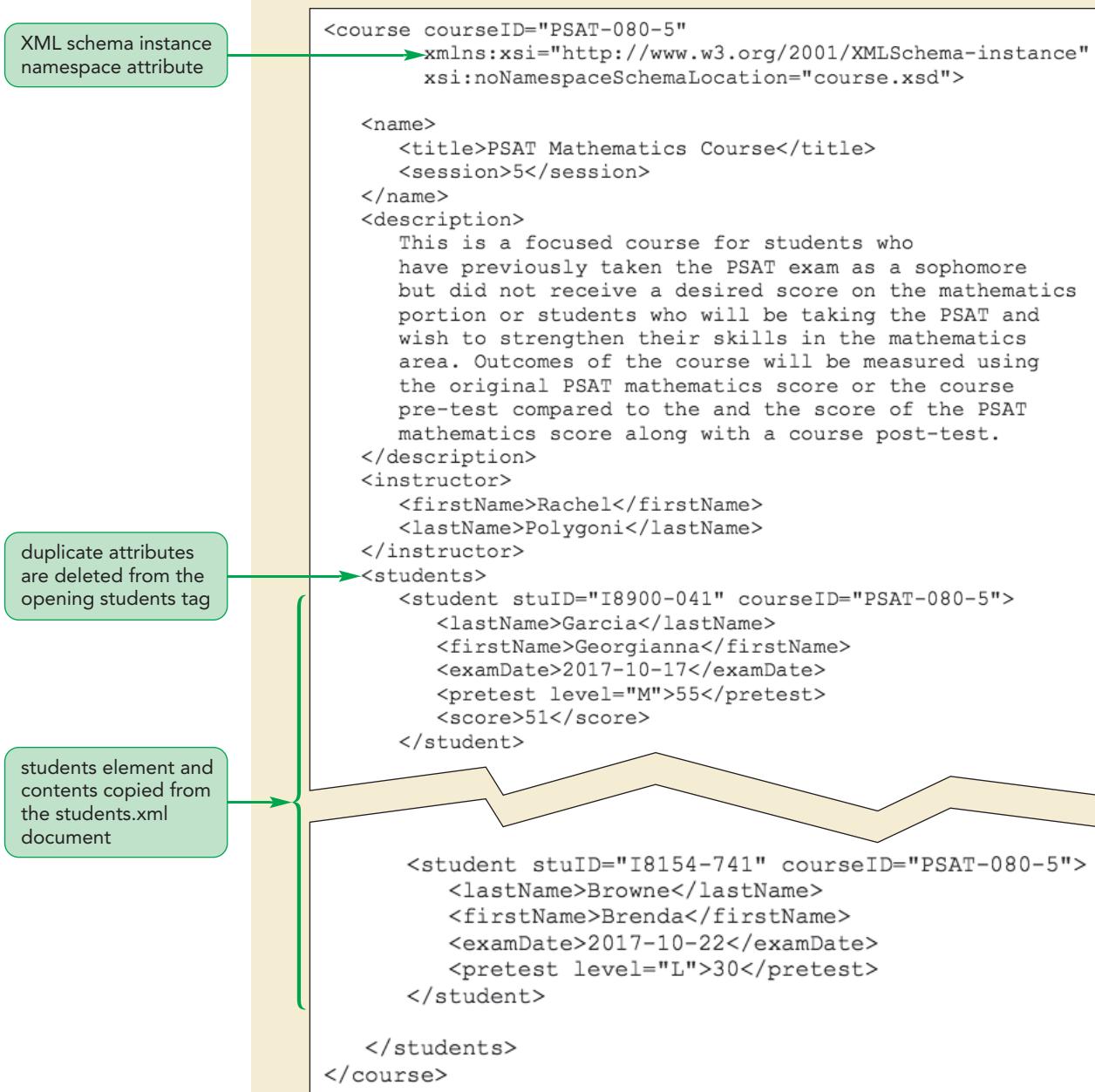
- 1. If necessary, return to the **psat.xml** file in your XML editor, and then save a copy of the file with the name **psatstudents.xml**.
- 2. In the comment section, change the filename to **psatstudents.xml**; in the supporting files list, add the filenames **course.css**, **students.css**, and **studentsvb.xsd**; and then save your changes.
- 3. Return to the **students.xml** file, and then copy the contents of the opening `<students>` tag through the closing `</students>` tag to the Clipboard.
- 4. Return to the **psatstudents.xml** file in your XML editor, and then insert the copied students.xml contents below the closing `</instructor>` tag.

Be sure to scroll down and copy all the student information in the students.xml file including the closing `</students>` tag, rather than stopping at one of the `</student>` tags.

5. In the opening <students> tag, delete both attributes and their values. The `xmlns` attribute declares the XML Schema instance namespace; however, it's unnecessary because the namespace is already declared in the `course` element, which encloses all the contents of this document. In addition, because you'll be using namespaces in this compound document, the `noNamespaceSchemaLocation` is no longer relevant. Figure 4-11 shows the compound document containing elements from both the course and students vocabularies.

Figure 4-11

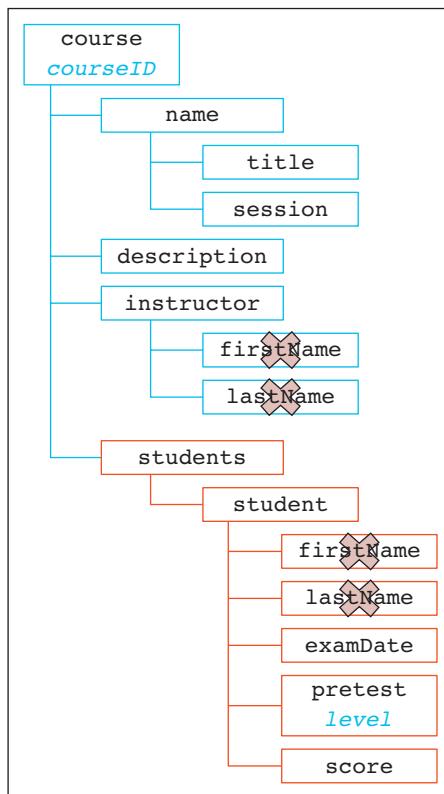
## Compound document showing information on first and last students



6. Save your changes to the `psatstudents.xml` file.

Notice that after combining the two vocabularies, some elements have the same names: Both the `instructor` and `student` elements contain child elements named `lastName` and `firstName`, as illustrated in Figure 4-12. Gabby would like you to investigate whether this is a problem.

**Figure 4-12** Name collision in a compound document



## Understanding Name Collision

The duplication of these element names is an example of name collision, which occurs when the same element name from different XML vocabularies is used within a compound document. Higher Ed Test Prep could have been more careful in choosing element names to prevent name collisions among different vocabularies. However, name collisions are often unavoidable. After all, one benefit of XML vocabularies is the ability to use simple element names to describe data. Creating complex element names to avoid name collisions eliminates this benefit. Moreover, there are other XML vocabularies such as XHTML over which Gabby has no control. XHTML element names such as `title` and `address` are certain to be found in thousands of XML vocabularies.

Gabby could also avoid combining elements from different vocabularies in the same document to prevent name collisions; however, this would make XML a poor information tool. Instead, Gabby can use namespaces to distinguish elements in one vocabulary from elements in another vocabulary.

## Working with Namespaces in an Instance Document

A namespace is a defined collection of element and attribute names. For example, the collection of element and attribute names from Gabby's courses vocabulary could make up a single namespace. Likewise, the element and attribute names from the students vocabulary could constitute a different namespace. Applying a namespace to an XML document involves two steps:

1. Declare the namespace.
2. Identify the elements and attributes within the document that belong to that namespace.

First, you will review how to declare a namespace.

### Declaring and Applying a Namespace to a Document

To declare and apply a namespace to a document, you add the attributes

```
xmlns="uri"
xsi:schemaLocation="uri schema"
```

to an element in the document, where *uri* is the URI of the namespace and *schema* is the location and name of the schema file. For example, the following code declares a namespace with the URI `http://example.com/higheredtestprep/course/ns` within the `course` element and applies the schema file `course.xsd` to the document:

```
<course courseID="PSAT-080-5"
    xmlns="http://example.com/higheredtestprep/course/ns"
    xsi:schemaLocation="http://example.com/higheredtestprep/
    course/ns course.xsd">
...
</course>
```

The number of namespace attributes that can be declared within an element is unlimited.

### REFERENCE

#### Declaring and Applying a Namespace in an Instance Document

- To declare a namespace, add the attribute

```
xmlns:prefix="uri"
```

to an element in the document, where *prefix* is the namespace prefix and *uri* is the URI of the namespace.

- To apply a schema file to a namespace you've declared, add the attribute

```
xsi:schemaLocation="uri schema"
```

where *schema* is the location and name of the schema file.

- To apply a namespace to an element, add the namespace prefix

```
<prefix:element> ... </prefix:element>
```

to the element's opening and closing tags, where *prefix* is the namespace prefix and *element* is the local part of the qualified element name. If no prefix is specified, the element is assumed to be part of the default namespace.

- To apply a namespace to an attribute, add the namespace prefix

```
<element prefix:attribute="value"> ... </element>
```

to the attribute name, where *attribute* is the attribute name. By default, an attribute is part of the namespace of its containing element.

Gabby wants you to create namespaces for the course and students vocabularies. You will declare each namespace in the root element of the content to which it applies; for the course vocabulary, this is the `course` element, and for the students vocabulary, this is the `students` element. The URLs for the two namespaces will be

`http://example.com/higheredtestprep/course/ns`  
`http://example.com/higheredtestprep/students/ns`

These URIs do not point to actual sites on the web, but they do provide unique URIs for the two namespaces.

### To declare the course and students namespaces:

- 1. If necessary, return to the `psatstudents.xml` file in your XML editor.
- 2. Within the opening `<course>` tag, delete the code  
`xsi:noNamespaceSchemaLocation="course.xsd"`, and then add the following code to declare the course namespace and specify the location of the schema file:  
`xmlns="http://example.com/higheredtestprep/course/ns"  
 xsi:schemaLocation="http://example.com/higheredtestprep/course/  
 ns course.xsd"`
- 3. Within the opening `<students>` tag, add the following code to declare the students namespace and specify the schema location:  
`xmlns="http://example.com/higheredtestprep/students/ns"  
 xsi:schemaLocation="http://example.com/higheredtestprep/  
 students/ns studentsvb.xsd"`

Compare your namespace declarations to those in Figure 4-13.

Figure 4-13

### Declaring namespaces within a compound document

The diagram shows the XML code for `psatstudents.xml` with annotations explaining the namespace declarations and schema locations.

```

<course courseID="PSAT-080-5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://example.com/higheredtestprep/course/ns"
    xsi:schemaLocation="http://example.com/higheredtestprep/course/ns course.xsd">

    <name>
        <title>PSAT Mathematics Course</title>
        <session>5</session>
    </name>
    <description>
        This is a focused course for students who have previously taken the PSAT exam as a sophomore but did not receive a desired score on the mathematics portion or students who will be taking the PSAT and wish to strengthen their skills in the mathematics area. Outcomes of the course will be measured using the original PSAT mathematics score compared to a course post-test, or a course pre-test compared to the subsequent PSAT mathematics score.
    </description>
    <instructor>
        <firstName>Rachel</firstName>
        <lastName>Polygoni</lastName>
    </instructor>
    <students xmlns="http://example.com/higheredtestprep/students/ns"
        xsi:schemaLocation="http://example.com/higheredtestprep/students/ns studentsvb.xsd">
        <student stuID="I8900-041" courseID="PSAT-080-5">
    </students>

```

Annotations in the diagram:

- A green callout labeled "course namespace declaration" points to the first line of the XML code: `<course courseID="PSAT-080-5"`.
- A green callout labeled "location of schema file for the course vocabulary" points to the schema location for the `course` namespace: `xsi:schemaLocation="http://example.com/higheredtestprep/course/ns course.xsd"`.
- A green callout labeled "students namespace declaration" points to the schema location for the `students` namespace: `xsi:schemaLocation="http://example.com/higheredtestprep/students/ns studentsvb.xsd"`.
- A green callout labeled "location of schema file for the students vocabulary" points to the schema location for the `students` namespace: `xsi:schemaLocation="http://example.com/higheredtestprep/students/ns studentsvb.xsd"`.

► 4. Save the changes to `psatstudents.xml`.

## Applying a Namespace to an Element

In an instance document containing elements from more than one namespace, after you declare the namespaces, you must indicate which elements in the document belong to each namespace. This process involves two steps:

1. Associate the namespace declaration with a prefix.
2. Add the prefix to the tags for each element in the namespace.

To apply an XML namespace to an element, you qualify the element's name. A qualified name, or qname, is an element name consisting of two parts—the namespace prefix that identifies the namespace, and the local part or local name that identifies the element or attribute within that namespace. The general form for applying a qualified name to a two-sided tag is

```
<prefix:element> ... </prefix:element>
```

where *prefix* is the namespace prefix and *element* is the local part. An element name without such a prefix is referred to as an unqualified name. You worked with qualified names previously when specifying elements from the XML Schema vocabulary in a schema file, using the `xs:` or `xsi:` prefix.

Namespaces have a scope associated with them. The scope of a namespace declaration declaring a prefix extends from the beginning of the opening tag to the end of the corresponding closing tag. The namespace declared in a parent element is connected with—or bound to—the defined prefix for that element as well as for all of its child elements. This is true unless the given prefix in the parent is overridden in a child element that has been assigned a different namespace. Some XML authors add all namespace declarations to the document's root element so that the namespace is available to all elements within the document. The association between the namespace and the prefix declared in an element does not apply to the siblings of that element.

A single namespace prefix can be declared as an attribute of an element, as shown in this example:

```
<crs:course courseID="PSAT-080-5"
    xmlns:crs="http://example.com/higheredtestprep/course/ns">
    <name>
        <title>PSAT Mathematics Course</title>
        <session>5</crs:session>
    </name>
    ...
</crs:course>
...
```

The opening `<course>` tag includes both the namespace prefix and the `xmlns` attribute to declare the namespace. This indicates that the `course` element itself is part of the namespace that it declares.

**INSIGHT**

### Qualified and Unqualified Names

The use of qnames in elements and attributes is controversial because it creates a dependency between the content of the document and its markup. However, in its official position, the W3C doesn't discourage this practice. The syntax for default namespaces was designed for convenience, but they tend to cause more confusion than they're worth. The confusion typically stems from the fact that elements and attributes are treated differently, and it's not immediately apparent that nested elements are being assigned the default namespace identifier. Nevertheless, in the end, choosing between prefixes and default namespaces is mostly a matter of style except when attributes come into play.

In XML Schema, any element or attribute with global scope must be entered as a qualified name (i.e., with a namespace prefix). The reason is that elements and attributes with global scope are attached to the schema's target namespace, while elements and attributes declared locally are not. In the instance document, this is reflected by qualifying those global elements or attributes.

This fact may affect your choice of schema designs. In a Flat Catalog, all elements and attributes are declared globally, so each element and attribute must be qualified in the instance document. Because Venetian Blind and Russian Doll designs have a single global element, only the root element must be qualified in the instance document.

In the psatstudents.xml document, you'll use the `crs` prefix for elements from the course namespace and the `stu` prefix for elements from the students namespace. You will apply each namespace to the parent element that belongs to that namespace. There is no need to assign attributes to namespaces, so you will not add prefixes to attributes in the psatstudents.xml file.

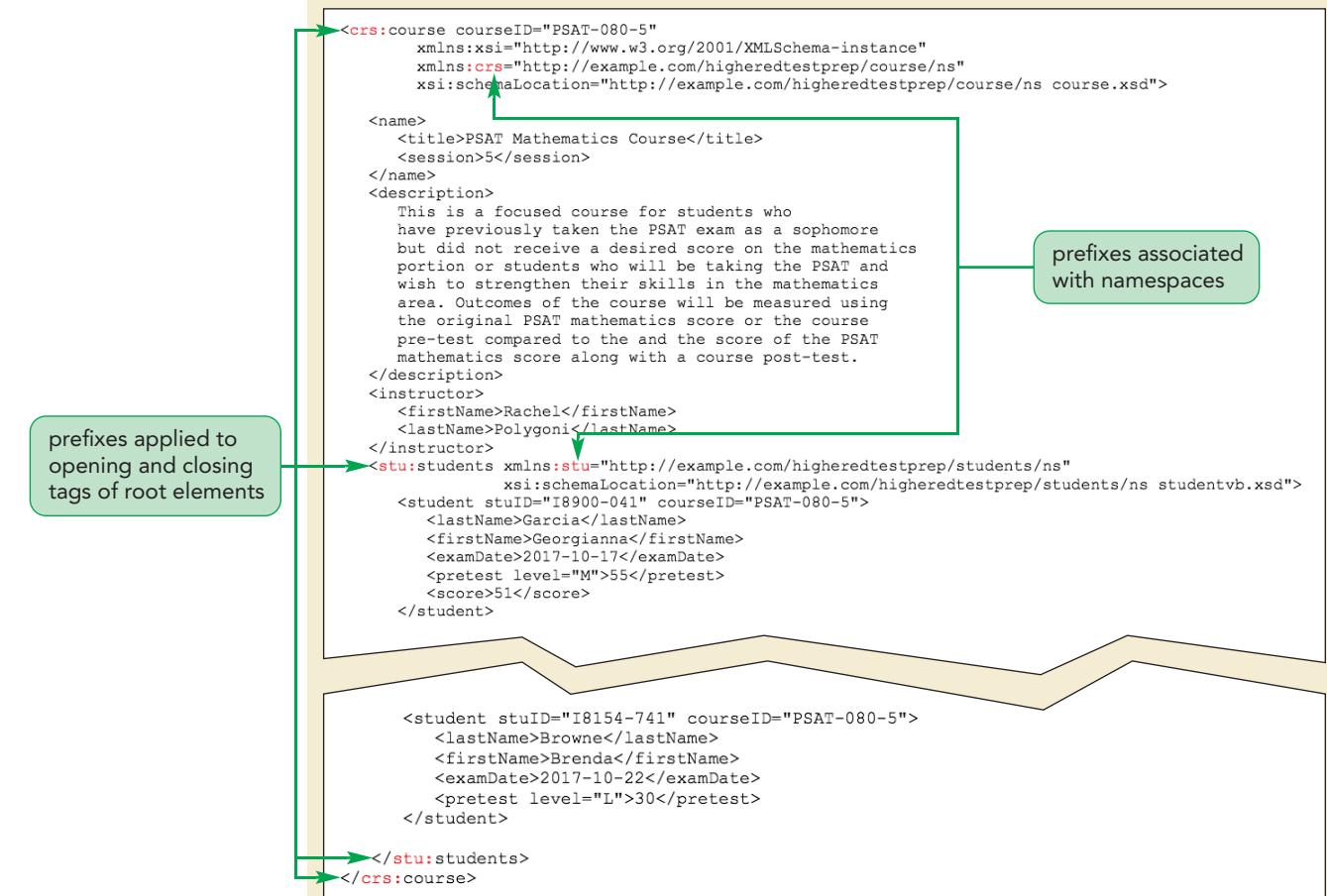
#### To apply the courses and students namespaces:

- 1. Verify that the **psatstudents.xml** file is the active file in your XML editor.
- 2. In the opening `<course>` tag, add the prefix `:crs` to the `xmlns` attribute that declares the course namespace so the attribute reads as follows:  
`xmlns:crs="http://example.com/higheredtestprep/course/ns"`  
This associates the `crs` prefix with the course namespace.
- 3. In the opening `<students>` tag, add the prefix `:stu` to the `xmlns` attribute that declares the students namespace so the attribute reads as follows:  
`xmlns:stu="http://example.com/higheredtestprep/students/ns"`  
This associates the `stu` prefix with the students namespace.
- 4. In the opening `<course>` tag, insert the `crs:` prefix just before the word `course`, and then repeat for the closing `</course>` tag.  
Adding the `crs` prefix to the opening and closing `<course>` tags specifies that the `course` element is part of the course namespace.
- 5. In the opening `<students>` tag, insert the `stu:` prefix just before the word `students`, and then repeat for the closing `</students>` tag. Figure 4-14 shows the updated code.

Add the `crs:` prefix to the closing tag as well as to the opening tag.

Figure 4-14

## Declaring namespaces within a compound document

**TIP**

By switching from the Flat Catalog to the Venetian Blind layout, you avoided having to change every element and attribute name in the instance document into a qualified name.

You don't have to qualify any of the child elements and attributes of the `course` element because none of them were declared globally in the `course.xsd` schema file. Only the `course` element was defined globally, which is why it is the only element that requires a qualified name. The same is true for the `students` element.

- 6. Save your changes to the **psatstudents.xml** file.

## Working with Attributes

Like an element name, an attribute can be qualified by adding a namespace prefix. The syntax to qualify an attribute is

```
<element prefix:attribute="value"> ... </element>
```

where `prefix` is the namespace prefix and `attribute` is the attribute name. For example, the following code uses the `crs:` prefix to assign both the `course` element and the `courseID` attribute to the same course namespace:

```
<crs:course crs:courseID="PSAT-080-5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:crs="http://example.com/higheredtestprep/course/ns">
...
</crs:course>
```

Unlike element names, there is no default namespace for attribute names. Default namespaces apply to elements, but not to attributes. An attribute name without a prefix is assumed to belong to the same namespace as the element that contains it. This means you could write the code listed above without the `crs:` prefix before the `courseID` attribute name and have the same result. In the code that follows, the `courseID` attribute is automatically assumed to belong to the course namespace, even though it lacks the `crs` prefix:

```
<crs:course courseID="PSAT-080-5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:crs="http://example.com/higheredtestprep/course/ns">
...
</crs:course>
```

Because an attribute is automatically associated with the namespace of its element, you rarely need to qualify an attribute name. The only exception occurs when an attribute from one namespace needs to be used in an element from another namespace. For example, XHTML uses the `class` attribute to associate elements belonging to a common group or class. You could attach the `class` attribute from the XHTML namespace to elements from other namespaces. Because the `class` attribute is often used in CSS to apply common formats to groups of elements, using the `class` attribute in other XML elements would apply this feature of CSS to those elements as well.

For Gabby's document, there is no need to assign attributes to namespaces, so you will not specify namespaces for the attributes in the `psatstudents.xml` file.

Now that you've specified namespaces for the root elements of the two vocabularies you're using, you'll use an XML validator to validate your compound document.

### To validate the compound document:

- 1. Use an XML validator to validate the `psatstudents.xml` document. The validator returns a number of errors.
- 2. Examine the text of the error messages for the reported errors. The first error message says that the target namespace of the schema document is 'null.'

**Trouble?** If you're using Exchanger XML Editor and your first error message doesn't say that the target namespace of the schema document is 'null', compare your code to Figures 4-13 and 4-14, fix any errors you find, and then revalidate until the target namespace error is the first error.

Gabby's compound document is invalid because even though the namespaces in the compound document are associated with URIs, the schemas themselves are not. For the instance document to be valid, you must add this information to each schema file associated with it.

## Associating a Schema with a Namespace

So far, you've specified the URI `http://example.com/higheredtestprep/course/ns` as the URI for the course vocabulary and the URI

`http://example.com/higheredtestprep/students/ns`

as the URI for the students vocabulary in the `psatstudents.xml` compound document. Next, you need to place the schemas themselves in the namespaces.

### Targeting a Namespace

To associate the rules of a schema with a namespace, you declare the namespace of the instance document in the schema element and then make that namespace the target of the schema using the `targetNamespace` attribute. The code to set the schema namespace is

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:prefix="uri"
            targetNamespace="uri">
    ...
</xs:schema>
```

where `prefix` is the prefix of the namespace and `uri` is the URI of the namespace. The `prefix` value is optional. You can omit it to make the namespace of the instance document the default namespace. For example, to associate Gabby's `studentsvb.xsd` schema file with the namespace of the students vocabulary, you would modify the schema element as follows:

#### TIP

This code uses the `xsi:` prefix for the XML Schema namespace because that's the prefix Gabby uses in her documents.

```
<xsi:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"
             xmlns="http://example.com/higheredtestprep/students/ns"
             targetNamespace="http://example.com/higheredtestprep/students/ns">
    ...
</xsi:schema>
```

Any customized data types, named types, elements, element groups, or attributes created in the schema are considered part of the target namespace. This allows you to make validation rules part of an XML vocabulary. For example, the `pretest` element is part of the students vocabulary, as is the rule that the `pretest` element must have a minimum value of 0 and a maximum value of 80.

#### REFERENCE

### Targeting a Namespace in a Schema and Applying the Namespace to an Instance Document

- To target a schema to a namespace, add the attributes

```
xmlns:prefix="uri"
targetNamespace="uri"
```

to the `schema` element, where `prefix` is the optional prefix of the namespace and `uri` is the URI of the namespace.

- To apply a schema to a document with a namespace, add the attributes

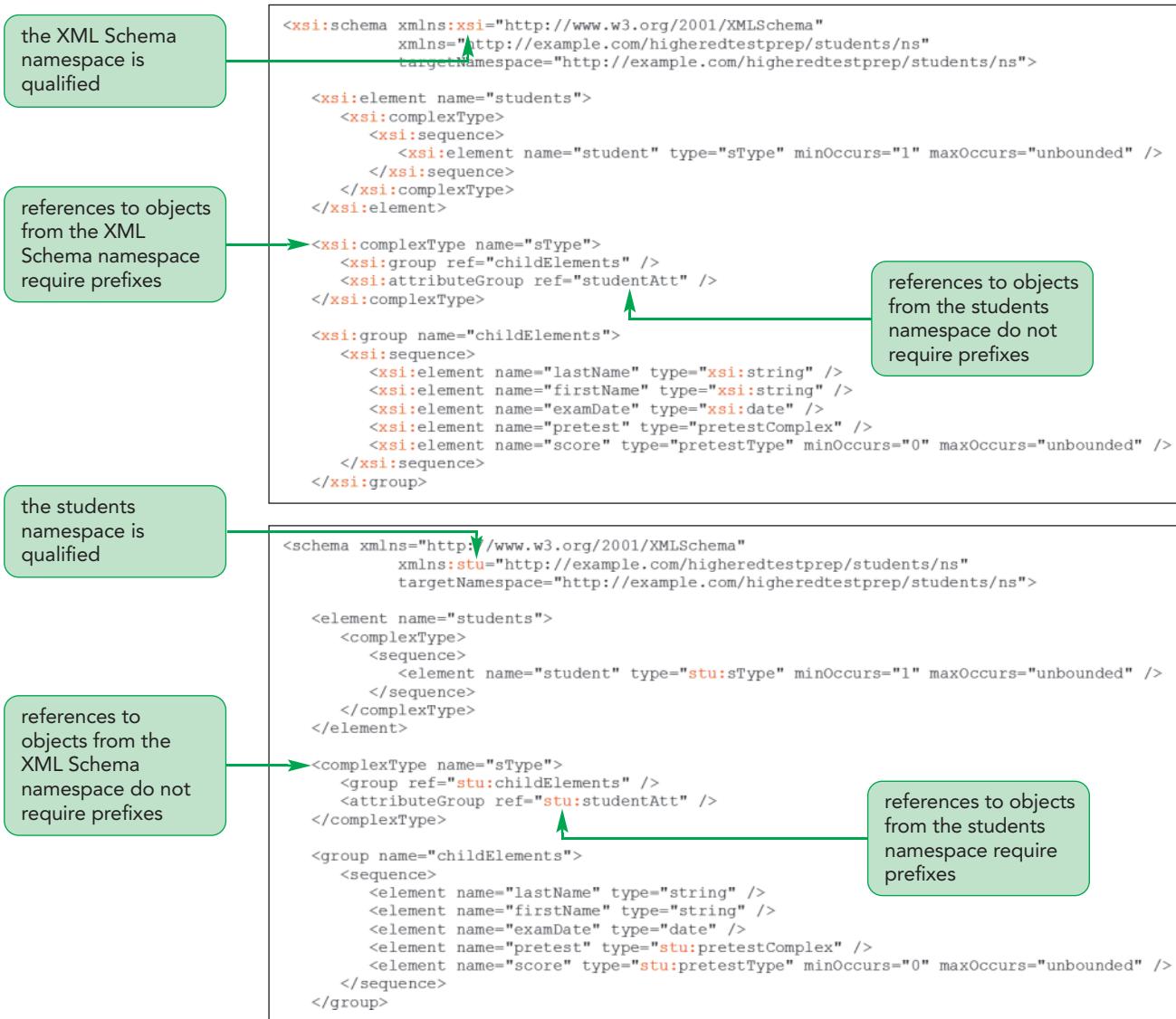
```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:prefix="uri"
xsi:schemaLocation="uri schema"
```

to the instance document's root element, where `prefix` is the namespace prefix, `uri` is the URI of the namespace, and `schema` is the schema file. All global elements and attributes declared in the schema must be qualified in the instance document.

If you use the vocabulary's namespace as the default namespace for the schema, you do not have to qualify any references to those customized objects. On the other hand, if you apply a prefix to the namespace, references to those objects must be qualified by that prefix. Figure 4-15 shows both possibilities—one in which references to objects from the XML Schema vocabulary are qualified, and the other in which XML Schema is the default namespace and references to customized objects are qualified.

Figure 4-15

### A schema with and without qualified XML Schema object names



You will modify Gabby's studentsvb.xsd schema file, using <http://example.com/higheredtestprep/students/ns> as the default and target namespace of the schema. You'll also modify the course.xsd schema file using <http://example.com/higheredtestprep/course/ns>.

**To associate each schema with a namespace:**

- 1. Return to the **studentsvb.xsd** document in your XML editor.
- 2. In the file, add the following attributes to the root schema element:

```
xmlns="http://example.com/higheredtestprep/students/ns"
targetNamespace="http://example.com/higheredtestprep/students/ns"
```

Figure 4-16 shows the attributes inserted in the code.

**Figure 4-16****Associating the students schema with a namespace**

```
<xsi:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"
             xmlns="http://example.com/higheredtestprep/students/ns"
             targetNamespace="http://example.com/higheredtestprep/students/ns">

    <xsi:element name="students">
```

- 3. Save your changes to the **studentsvb.xsd** file.
  - 4. In the **course.xsd** file, add the following attributes to the root schema element:
- ```
xmlns="http://example.com/higheredtestprep/course/ns"
targetNamespace="http://example.com/higheredtestprep/course/ns"
```

Figure 4-17 shows the attributes inserted in the code.

**Figure 4-17****Associating the course schema with a namespace**

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://example.com/higheredtestprep/course/ns"
            targetNamespace="http://example.com/higheredtestprep/course/ns">

    <xs:element name="course">
```

- 5. Save your changes to the **course.xsd** file, return to **psatstudents.xml** in your XML validator, and then validate the document. The document still fails validation, this time with a message about **stu:students** not being expected as a child element.

**Trouble?** If you receive any validation errors other than the error described in Step 5, compare your code against the preceding figures, fix any errors you find, and then revalidate until you receive only the error described in Step 5.

Gabby is pleased that you've been able to combine both the course and the students namespaces in a single instance document. Your final step will be to combine the schema information for both namespaces in the **course.xsd** schema document. XML Schema includes two methods for achieving this—including and importing schemas.

## Including and Importing Schemas

You include a schema file when you want to combine schema files from the same namespace. This might be the case if one schema file contains a collection of customized data types that you want shared among many different files, and another schema file contains a collection of elements and attributes that define the structure of a particular document. To include a schema, you add the element

```
<xsi:include schemaLocation="schema" />
```

as a child of the root `schema` element, where `schema` is the name of the schema file to be included. The effect is to combine the two schema files into a single schema that can then be applied to a specific instance document. In an environment in which large and complex XML vocabularies are developed, different teams might work on different parts of the schema, using the `include` element to combine the different parts into a finished product. Rather than having one large and complex schema file, you can break the schema into smaller, more manageable files that can be shared and combined.

The other way to combine schemas is through importing, which is used when the schemas come from different namespaces. The syntax of the `import` element is

```
<xsi:import namespace="uri" schemaLocation="schema" />
```

where `uri` is the URI of the namespace for the imported schema and `schema` is again the name of the schema file. For example, to import the contents of the students schema into Gabby's course schema, you would add the following `import` element to the `course.xsd` schema file:

```
<xsi:import  
    namespace="http://example.com/higheredtestprep/students/ns"  
    schemaLocation="studentsvb.xsd" />
```

A schema can contain any number of `include` and `import` elements. Each must be globally declared as a direct child of the root `schema` element.

### REFERENCE

#### Including and Importing Schemas

- To combine schemas from the same namespace, add the element

```
<xsi:include schemaLocation="schema" />
```

as a child of the `schema` element, where `schema` is the name of the schema file.

- To combine schemas from different namespaces, use

```
<xsi:import namespace="uri" schemaLocation="schema" />
```

where `uri` is the URI of the imported schema's namespace and `schema` is the name of the schema file.

You will use the `import` element to import the `studentsvb.xsd` schema file into the `course.xsd` file.

### To import the `studentsvb.xsd` schema file:

- 1. Return to the `course.xsd` file in your XML editor.
- 2. Directly below the opening tag of the `schema` element, insert the following `import` element:

```
<xs:import
    namespace="http://example.com/higheredtestprep/students/ns"
    schemaLocation="studentsvb.xsd" />
```

Figure 4-18 shows the `import` element inserted in the code.

**Figure 4-18**

### Importing a schema file

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/course/ns"
    targetNamespace="http://example.com/higheredtestprep/course/ns">

    <xs:import namespace="http://example.com/higheredtestprep/students/ns"
        schemaLocation="studentsvb.xsd" />
    <xs:element name="course">
```

This element assigns the namespace  
`http://example.com/higheredtestprep/students/ns`  
 to the contents of the `studentsvb.xsd` schema document.

- 3. Save your work, validate the file, and then if necessary troubleshoot any validation errors until the file validates.

## Referencing Objects from Other Schemas

After a schema is imported into another schema file, any objects it contains with global scope can be referenced in that file. To reference an object from an imported schema, you must declare the namespace of the imported schema in the `schema` element. You can then reference the object using the `ref` attribute or the `type` attribute for customized simple and complex types.

Gabby wants the `students` element to be placed directly after the `instructor` element in this schema, to match the location where you placed the `students` element and its content in the instance document. You will add the reference to the `students` element to the schema now.

### TIP

When referencing elements in an imported schema file, the prefix does not have to match the prefix used in the imported schema file.

### To reference the `students` element in the `course.xsd` file:

- 1. In the `course.xsd` file, add the following namespace declaration to the root `schema` element:
 

```
xmlns:stu="http://example.com/higheredtestprep/students/ns"
```
- 2. Insert the following element reference directly below the closing `</xs:element>` tag for the `instructor` element declaration:
 

```
<xs:element ref="stu:students" />
```

This code tells validators that in the sequence of elements within the `course` element, the `students` element from the `students` namespace should follow the `instructor` element from the `course` namespace. The element reference is qualified with a namespace prefix to indicate to validators that this reference

points to a global object found in the students namespace. Figure 4-19 shows the revised schema code.

Figure 4-19

### The course and students schemas combined in a single file

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/course/ns"
    targetNamespace="http://example.com/higheredtestprep/course/ns"
    xmlns:stu="http://example.com/higheredtestprep/students/ns">

    <xs:import namespace="http://example.com/higheredtestprep/students/ns"
        schemaLocation="studentsvb.xsd" />
    <xs:element name="course">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="title" type="xs:string" />
                            <xs:element name="session" type="xs:string" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="description" type="xs:string" />
                <xs:element name="instructor">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="firstName" type="xs:string" />
                            <xs:element name="lastName" type="xs:string" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element ref="stu:students" />
            </xs:sequence>
            <xs:attribute name="courseID" type="xs:ID" />
        </xs:complexType>
    </xs:element>

</xs:schema>
```

- 3. Save the changes to the **course.xsd** document.
- 4. Return to **psatstudents.xml** in your XML editor.
- 5. Validate the XML content against the schema. The document passes validation, with the validator drawing rules for content and structure from the two different schema files.

**Trouble?** If your psatstudents.xml document doesn't validate, compare your course.xsd file to Figure 4-19, and then edit your code as necessary until the psatstudents.xml document validates.

This example provides a glimpse of the power and flexibility of schemas in working with multiple vocabularies and namespaces. In more advanced applications, large schema structures can be created to validate equally complex XML environments involving dozens of documents and vocabularies. The XML Schema language is also flexible enough to provide control over which elements and attributes are validated, and how they are validated.

Gabby is pleased with the work you have done on creating a schema for the compound document describing the features of the PSAT Mathematics Course and the list of students enrolled in that course. She'll use the document as a model for creating compound documents for other courses.



### Problem Solving: To Namespace or Not to Namespace?

XML documents can have any format unless specifically tied to a vocabulary. The question of whether or not to namespace often arises. Because namespaces must be added to both the XML document and any associated CSS, adding a namespace prefix requires quite a bit of document customization. Some programmers feel that using namespaces in XML and CSS documents “clutters” the code, and they argue that it would be better to modify any custom vocabularies as much as possible to avoid name collision problems. This would allow the XML and CSS documents to remain more flexible. To avoid namespace collisions, the name of one item (typically the one used less often) would need to be changed to some other name. Although this seems like a simple solution, it could be difficult to implement because there is no master list of all element and attribute names for XML vocabularies. Therefore, you may not always be able to predetermine where every possible name collision will occur. Another approach would be to put unique characters before the names so that the names differ and further name collisions are unlikely to happen. Regardless of which approach you take to avoid namespace collisions, it should be applied consistently throughout the system.

## Combining Standard Vocabularies

So far you've worked only with the custom XML vocabularies that Gabby has created for Higher Ed Test Prep. The standard vocabularies that are shared throughout the world, such as XHTML, RSS, and MathML, can also be combined within a single compound document. Many of these standard vocabularies have unique URIs, some of which are listed in Figure 4-20.

Figure 4-20

Namespace URIs for standard vocabularies

Vocabulary	Namespace URI
CML	<a href="http://www.xml-cml.org/schema">http://www.xml-cml.org/schema</a>
MathML	<a href="http://www.w3.org/1998/Math/MathML">http://www.w3.org/1998/Math/MathML</a>
iTunes Podcast	<a href="http://www.itunes.com/dtds/podcast-1.0.dtd">http://www.itunes.com/dtds/podcast-1.0.dtd</a>
SMIL	<a href="http://www.w3.org/2001/SMIL20/Language">http://www.w3.org/2001/SMIL20/Language</a>
SVG	<a href="http://www.w3.org/2000/svg">http://www.w3.org/2000/svg</a>
VoiceML	<a href="http://www.w3.org/2001/vxml">http://www.w3.org/2001/vxml</a>
XForms	<a href="http://www.w3.org/2002/xforms">http://www.w3.org/2002/xforms</a>
XHTML	<a href="http://www.w3.org/1999/xhtml">http://www.w3.org/1999/xhtml</a>

### TIP

Internet Explorer versions before IE9 support the combination of MathML with other languages only if an add-in is installed.

As XML has developed as a standard language for sharing markup data, web browsers have extended and improved their ability to support documents that combine multiple vocabularies. For example, current versions of Internet Explorer, Firefox, Chrome, Safari, and Opera support documents that combine both the XHTML and MathML languages.

**INSIGHT**

### Compound Documents and Podcasting

Podcasting is an area where compound documents are used. Information about the location and content of podcasts is written in the XML vocabulary language RSS. As you learned in an earlier tutorial, RSS is used for syndicating text, video, or audio content. However, if you want to list your podcast on Apple's iTunes Music Store to make it more accessible to the general population, you must add elements that are specific to the needs of iTunes but that are not part of RSS. Therefore, the final podcast document contains elements from both RSS and the iTunes vocabulary.

To declare the iTunes namespace, you add the following attribute to the root `rss` element of the podcast document:

```
<rss version="2.0"
  xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd">
```

After you have declared the iTunes namespace, you can populate the rest of the document with iTunes-specific elements. The following text shows a portion of a compound document describing a podcast channel using elements from both RSS and iTunes:

```
<channel>
  <title>Jazz Pod Sessions</title>
  <link>http://example.com</link>
  <description>Enjoy jazz music from JPS</description>
  <itunes:author>David Hmong</itunes:author>
  <itunes:category text="Music">
    <itunes:category text="Jazz" />
  </itunes:category>
  ...
</channel>
```

The iTunes-specific elements listed here—author and category—will be displayed in Apple's iTunes Music Store, providing additional information to potential subscribers of the feed. To augment the descriptions of individual episodes, you add iTunes-specific elements to each `<item>` tag in the RSS document. The following shows part of the code for one episode of a podcast:

```
<item>
  <title>Jazz at Carnegie Hall</title>
  <itunes:subtitle>Famous Concerts</itunes:subtitle>
  <itunes:summary>Jazz from Carnegie Hall</itunes:summary>
  <itunes:author>Various</itunes:author>
  <itunes:duration>59:23</itunes:duration>
  ...
</item>
```

This particular episode highlights famous jazz concerts at Carnegie Hall. The code uses iTunes-specific elements to provide a subtitle for the show, a summary, the show's author, and the duration of the show in minutes and seconds.

The iTunes elements listed here represent only a fraction of the elements you can add to podcast code. You can learn more about podcasting and how to write compound documents involving both RSS and iTunes by visiting Apple's website.

**REVIEW****Session 4.2 Quick Check**

1. What is a name collision?
2. How do namespaces prevent the problem of name collisions?
3. If an attribute name is unqualified, what namespace is it presumed to belong to?
4. How does importing a schema file differ from including a schema file, with respect to the namespace of the schema?
5. How do you reference an object with global scope from an imported schema?

# Session 4.3 Visual Overview:

To link multiple style sheets to a compound document, you add a separate processing instruction for each style sheet.

Elements in an instance document must be qualified to be styled by linked style sheet rules.

```
<?xml-stylesheet type="text/css" href="students.css" ?>
<?xml-stylesheet type="text/css" href="course.css" ?>

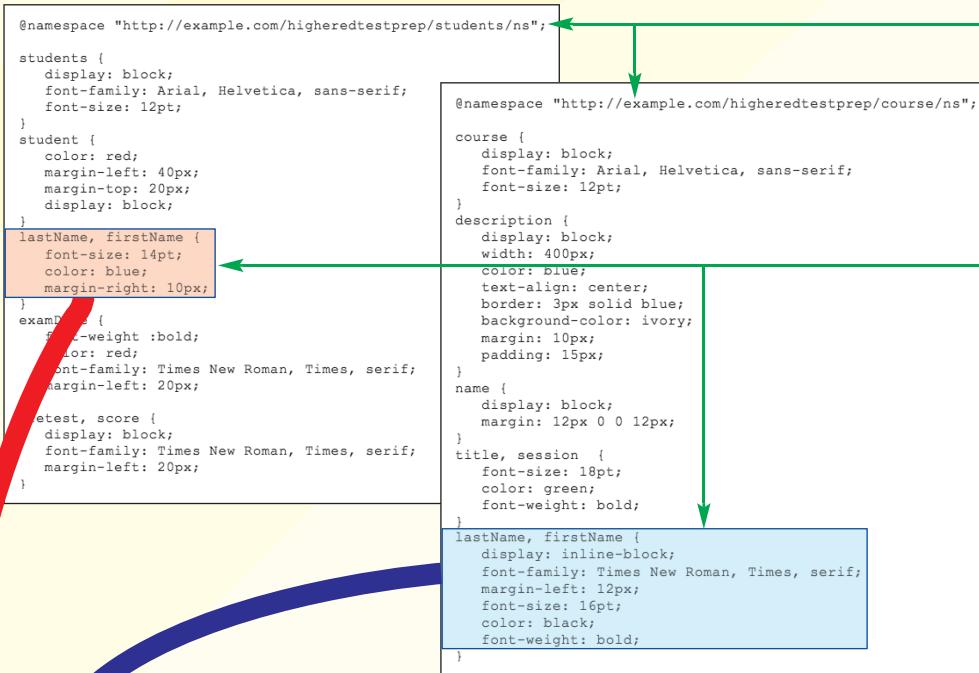
<crs:course courseID="PSAT-080-5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:crs="http://example.com/higheredtestprep/course/ns"
    xsi:schemaLocation="http://example.com/higheredtestprep/course/ns course.xsd">

    <name>
        <title>PSAT Mathematics Course</title>
        <session>5</session>
    </name>
    <description>
        This is a focused course for students who
        have previously taken the PSAT exam as a sophomore
        but did not receive a desired score on the mathematics
        portion or students who will be taking the PSAT and
        wish to strengthen their skills in the mathematics
        area. Outcomes of the course will be measured using
        the original PSAT mathematics score or the course
        pre-test compared to the and the score of the PSAT
        mathematics score along with a course post-test.
    </description>
    <instructor>
        <crs:firstName>Rachel</crs:firstName>
        <crs:lastName>Polygoni</crs:lastName>
    </instructor>
    <stu:students xmlns:stu="http://example.com/higheredtestprep/students/ns"
        xsi:schemaLocation="http://example.com/higheredtestprep/students/ns studentsvb.xsd">
        <student stuID="I8900-041" courseID="PSAT-080-5">
            <stu:lastName>Garcia</stu:lastName>
            <stu:firstName>Georgianna</stu:firstName>
            <examDate>2017-10-17</examDate>
            <pretest level="M">55</pretest>
            <score>51</score>
        </student>
        ...
    </stu:students>
</crs:course>
```

The lastName and firstName elements in the course vocabulary are qualified with the crs: prefix and formatted by the style sheet for the course vocabulary.

The lastName and firstName elements in the students vocabulary are qualified with the stu: prefix and formatted by the style sheet for the students vocabulary.

# Styling a Compound Document



You use an @namespace rule to declare a namespace in a style sheet.

A name collision occurs when two vocabularies are used in the same document, and the vocabularies contain one or more elements with the same name. These documents avoid name collisions in styling elements by specifying a default namespace for each style sheet.

**PSAT Mathematics Course 5**

This is a focused course for students who have previously taken the PSAT exam as a sophomore but did not receive a desired score on the mathematics portion or students who will be taking the PSAT and wish to strengthen their skills in the mathematics area. Outcomes of the course will be measured using the original PSAT mathematics score compared to a course post-test, or a course pre-test compared to the subsequent PSAT mathematics score.

Student Name	Score	Date
Rachel Polygoni		
Garcia Georgianna	55 51	2017-10-17
Smith Ryan	25	2017-10-17
Zheng Paddy	65	2017-10-17
Steinke Devon	51	2017-10-22
Browne Brenda	30	2017-10-22

Using namespaces in style sheets enables you to apply different styles to elements with the same name in different namespaces.

## Adding a Namespace to a Style Sheet

In the previous session, you added namespaces to Gabby's new psatstudents.xml compound document. To display the contents of this compound document, Gabby wants you to use styles from style sheets that she already uses for the students and course vocabularies. You'll link these files now to the psatstudents.xml compound document.

### To link the **students.css** and **course.css** style sheets to the **psatstudents.xml** file:

- 1. Use your XML editor to open **studentstxt.css** and **coursetxt.css** from the **xml04 ▶ tutorial** folder, enter **your name** and **today's date** in the comment section, and then save the files as **students.css** and **course.css**, respectively, in the same folder.
- 2. Examine the contents of the **students.css** and **course.css** files.
- 3. If you took a break after the previous session, make sure the **psatstudents.xml** document from the **xml04 ▶ tutorial** folder is open in your XML editor.
- 4. In the **psatstudents.xml** file, immediately below the comment section, enter the following two processing instructions to link the **students.css** and **course.css** style sheets to the **psatstudents.xml** document:

```
<?xmlstylesheet type="text/css" href="students.css" ?>
<?xmlstylesheet type="text/css" href="course.css" ?>
```

Figure 4-21 shows the processing instructions inserted in the code.

Figure 4-21

### Linking the instance document to the style sheets

```
Filename:          psatstudents.xml
Supporting Files: course.css, course.xsd, students.css, studentvb.xsd
-->
<?xmlstylesheet type="text/css" href="students.css" ?>
<?xmlstylesheet type="text/css" href="course.css" ?>

<crs:course courseID="PSAT-080-5"
```

- 5. Save your changes to the **psatstudents.xml** file, and then open the **psatstudents.xml** file in your browser. See Figure 4-22.

Figure 4-22

Compound document with default style sheets applied

This is a focused course for students who have previously taken the PSAT exam as a sophomore but did not receive a desired score on the mathematics portion or students who will be taking the PSAT and wish to strengthen their skills in the mathematics area. Outcomes of the course will be measured using the original PSAT mathematics score or the course pre-test compared to the and the score of the PSAT mathematics score along with a course post-test.

**Rachel Polygoni**

**Garcia Georgianna 2017-10-17**

**Smith Ryan 2017-10-17**

55  
51  
25

instructor and student names formatted with the same font color and size

**Trouble?** If the instructor name and the student names are displayed in blue instead of black in your browser, you entered the link to the course.css style sheet before the link to the students.css style sheet. To make your code match the figure, edit the code you just added to the instance document so the line referencing students.css is first, followed by the line referencing course.css, as in Figure 4-21.

The first and last names of the instructor and the students are displayed in the same color, size, and font. Instead, Gabby wants the instructor's name to be visually distinct from the students' names, as shown in the rendered document in Visual Overview 4.3.

The document's appearance is different than expected because the rules in the style sheet documents don't take into account the namespaces of the different elements in the instance document. Your next task is to add namespace support to the style sheets.

Recall that to apply a CSS style to an XML element, you use the style declaration

```
selector {attribute1:value1; attribute2:value2; ...}
```

where *selector* references an element or elements in the XML document. So, to set the width of the **student** element, you could enter the following style declaration:

```
student {width: 150px}
```

If an element has a qualified name such as *stu:student*, you do *not* include the prefix in the selector name, as follows:

#### Invalid code

```
stu:student {width: 150px}
```

This doesn't work with style sheets because CSS reserves the colon character for pseudo-elements and pseudo-classes. Instead, you must declare a namespace in the style sheet and then reference that namespace in the selector using a different syntax.

## Declaring a Namespace in a Style Sheet

To declare a namespace in a style sheet, you add the rule

```
@namespace prefix "uri";
```

to the CSS style sheet, where *prefix* is the namespace prefix and *uri* is the URI of the namespace. Both the prefix and the URI must match the prefix and URI used in the XML document. So, to declare the students namespace in Gabby's students.css style sheet, you would add the following rule:

```
@namespace stu "http://example.com/higheredtestprep/students/ns";
```

### TIP

If a namespace prefix is declared more than once, only the last instance is used in the style sheet.

Note that the prefix (*stu*) and the URI

`http://example.com/higheredtestprep/students/ns`

match the prefix and URI you entered in the previous session.

As with XML documents, the namespace prefix is optional. If the namespace prefix is omitted, the URI in the `@namespace` rule is considered to be the default namespace for the selectors in the style sheet. Any `@namespace` rules in the style sheet must come after all `@import` and `@charset` rules, and before any style declarations.

### INSIGHT

### Applying a Namespace to a Selector

After you have declared a namespace in a style sheet, you can associate selectors with that namespace by adding the namespace prefix to each selector name separated with the | symbol, as follows:

```
prefix|selector {attribute1: value1; attribute2: value2; ...}
```

For example, the style declaration

```
stu|lastname {width: 150px}
```

applies a width value of 150px to all `lastname` elements that belong to the `stu` namespace. You can also use the wildcard symbol (\*) to apply a style to any element within a namespace or to elements across different namespaces. For example, the style declaration

```
stu|* {font-size: 12pt}
```

applies the specified `font-size` value to any element within the `stu` namespace. Similarly, the declaration

```
*|student {width: 150px}
```

sets a width of 150 pixels for any element named `student` from any namespace. If you omit the namespace prefix from a selector, its style is also applied to all namespaces. For example, the declaration

```
student {width: 150px}
```

applies to all elements named `student` in any namespace.

In the `psatstudents.xml` instance document, the namespace collisions are occurring between the `students` and `course` vocabularies. As a result, styles intended for the `firstName` and `lastName` elements from one vocabulary are being applied to those elements in the other vocabulary as well. You'll add code to each style sheet now to associate it with a namespace.

## REFERENCE

### Declaring and Applying a Namespace in a CSS Style Sheet

- To declare a namespace in a CSS style sheet, add the rule

```
@namespace prefix "uri";
```

before any style declarations, where *prefix* is the namespace prefix and *uri* is the namespace URI. If no prefix is specified, the namespace URI is the default namespace for selectors in the style sheet.

- To apply a namespace to a selector, use the form

```
prefix|selector {attribute1: value1; attribute2: value2; ...}
```

where *prefix* is the namespace prefix and *selector* is a selector for an element or group of elements in the document.

You will add @namespace rules to the style sheets in the students.css and course.css files now.

### To declare and apply namespaces in the **students.css** and **course.css** style sheets:

- In the **students.css** style sheet, directly after the comment section, insert the following namespace declaration:

```
@namespace "http://example.com/higheredtestprep/students/ns";
```

Figure 4-23 shows the namespace declaration inserted in the code.

Figure 4-23

### Default namespace declared in **students.css** style sheet

```
/*
@namespace "http://example.com/higheredtestprep/students/ns";

students {
    display: block;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 12pt;
}
```

This code specifies http://example.com/higheredtestprep/students/ns as the default namespace for all selectors in the students.css document.

- Save your changes to the file.
- In the **course.css** style sheet in your XML editor, directly after the comment section, insert the following namespace declaration:

```
@namespace "http://example.com/higheredtestprep/course/ns";
```

This code specifies http://example.com/higheredtestprep/course/ns as the default namespace for all selectors in the course.css document. Figure 4-24 shows the namespace declaration inserted in the code.

Figure 4-24

**Default namespace declared in course.css style sheet**

```
/*
@namespace "http://example.com/higheredtestprep/course/ns";

course {
    display: block;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 12pt;
}
```

- 4. Save your changes to the **course.css** style sheet, and then reload **psatstudents.xml** in your browser. As shown in Figure 4-25, none of the text from either namespace is formatted.

Figure 4-25

**Unformatted document in browser after declaring style sheet namespaces**

PSAT Mathematics Course 5 This is a focused course for students who have previously taken the PSAT exam as a sophomore but did not receive a desired score on the mathematics portion or students who will be taking the PSAT and wish to strengthen their skills in the mathematics area. Outcomes of the course will be measured using the original PSAT mathematics score or the course pre-test compared to the and the score of the PSAT mathematics score along with a course post-test. Rachel Polygona Garcia Georgianna 2017-10-17 55 51 Smith Ryan 2017-10-17 25 Zheng Paddy 2017-10-17 65 Steinke Devon 2017-10-22 51 Browne Brenda 2017-10-22 30

You need to make a couple more changes to your code to enable both schemas and style sheets to work together in your instance document.

As you saw earlier in the tutorial, parsers can associate child elements with namespaces and validate those elements against schemas, based only on a prefix designated for the parent element. However, when browsers apply CSS styles to a default namespace, such as those you declared for the students.css and course.css style sheets, the browsers expect the name of each element being styled to be qualified. Because all of the elements that you want to style in your instance document are currently inheriting their namespace designations from their parent elements, rather than being qualified themselves, the browser didn't associate any of the styles in the style sheets with elements in the instance document. You'll make two changes in your code to make the elements styleable.

## Qualifying Elements and Attributes by Default

### TIP

The purpose of a schema is to describe a vocabulary, in which top-level nodes belong to their target namespace. For this reason, it is forbidden to define global elements that are unqualified when a target namespace is declared.

You can force all elements and attributes to be qualified, regardless of their scope, by adding the `elementFormDefault` and `attributeFormDefault` attributes

```
<xs:schema
    elementFormDefault="qualify"
    attributeFormDefault="qualify">
    ...
</xs:schema>
```

to the root `schema` element in the schema file, where `qualify` is either `qualified` or `unqualified`, specifying whether all the elements and attributes of the instance document must be qualified. The default value of both of these attributes is `unqualified` except for globally defined elements and attributes, which must always be qualified. To require all elements to be qualified but not all attributes (other than globally declared attributes), you enter the following code into the `schema` element:

```
<xs:schema
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    ...
</xs:schema>
```

### TIP

Many XML developers consider it a best practice to qualify all elements in the instance document to avoid confusion about which elements belong to which namespaces.

This is a common setup when you want to explicitly qualify each element name with a namespace prefix.

You can also set the qualification for individual elements or attributes by applying the `form` attribute

```
<xs:element name="name" form="qualified" />
<xs:attribute name="name" form="qualified" />
```

to the definitions in the schema, where `qualify` is again either `qualified` or `unqualified`. For example, the element declaration

```
<xs:element name="student" form="qualified" />
```

requires the `student` element to be qualified in the instance document, whether it has been declared globally or locally in the schema.

Browsers are looking for qualified element names in your instance document, but your schema is configured to expect unqualified names. To make your instance document work with both schemas and style sheets, you'll first add code to the `course.xsd` schema document to specify that all elements should be qualified. Then in your instance document, you'll make all elements qualified by adding prefixes to all elements.

**To specify that elements should be qualified and then qualify all elements:**

- 1. In the **course.xsd** schema file, within the opening `<schema>` tag, insert the following attributes, and then save your changes:

```
elementFormDefault="qualified"
attributeFormDefault="unqualified"
```

Figure 4-26 shows the attributes inserted in the code.

Figure 4-26

**Attributes added to course.xsd schema file**

```
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/course/ns"
    targetNamespace="http://example.com/higheredtestprep/course/ns"
    xmlns:stu="http://example.com/higheredtestprep/students/ns"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

<xss:import namespace="http://example.com/higheredtestprep/students/ns"
    schemaLocation="studentsvb.xsd" />
```

This code specifies that the schema expects all elements in an instance document to be qualified, and that it does not expect attributes in an instance document to be qualified.

- 2. In the **studentsvb.xsd** schema file, within the opening `<schema>` tag, insert the following attributes, and then save your changes:

```
elementFormDefault="qualified"
attributeFormDefault="unqualified"
```

Figure 4-27 shows the attributes inserted in the code.

Figure 4-27

**Attributes added to studentsvb.xsd schema file**

```
<xsi:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"
    xmlns="http://example.com/higheredtestprep/students/ns"
    targetNamespace="http://example.com/higheredtestprep/students/ns"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

<xsi:element name="students">
```

- 3. In the **psatstudents.xml** instance document, add the **stu:** prefix to the opening and closing tags of all elements nested within the **students** element, add the **crs:** prefix to the opening and closing tags of all elements within the root **course** element that are not already qualified with the **stu:** prefix, and then save your changes. See Figure 4-28.

Figure 4-28

## Namespace prefixes added to elements in compound document

crs: prefix added to opening and closing tags for all elements in the course namespace

stu: prefix added to opening and closing tags for all elements in the students namespace

```

<crs:course courseID="PSAT-080-5"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:crs="http://example.com/higheredtestprep/course/ns"
  xsi:schemaLocation="http://example.com/higheredtestprep/course/ns course.xsd">

  <crs:name>
    <crs:title>PSAT Mathematics Course</crs:title>
    <crs:session>5</crs:session>
  </crs:name>
  <crs:description>
    This is a focused course for students who
    have previously taken the PSAT exam as a sophomore
    but did not receive a desired score on the mathematics
    portion or students who will be taking the PSAT and
    wish to strengthen their skills in the mathematics
    area. Outcomes of the course will be measured using
    the original PSAT mathematics score or the course
    pre-test compared to the and the score of the PSAT
    mathematics score along with a course post-test.
  </crs:description>
  <crs:instructor>
    <crs:firstName>Rachel</crs:firstName>
    <crs:lastName>Polygoni</crs:lastName>
  </crs:instructor>

  <stu:students xmlns:stu="http://example.com/higheredtestprep/students/ns"
    xsi:schemaLocation="http://example.com/higheredtestprep/students/ns studentsvb.xsd">
    <stu:student stuID="I8900-041" courseID="PSAT-080-5">
      <stu:lastName>Garcia</stu:lastName>
      <stu:firstName>Georgianna</stu:firstName>
      <stu:examDate>2017-10-17</stu:examDate>
      <stu:pretest level="M">55</stu:pretest>
      <stu:score>51</stu:score>
    </stu:student>

    <stu:student stuID="I7711-121" courseID="PSAT-080-5">
      <stu:lastName>Smith</stu:lastName>
      <stu:firstName>Ryan</stu:firstName>
      <stu:examDate>2017-10-17</stu:examDate>
      <stu:pretest level="L">25</stu:pretest>
    </stu:student>

    <stu:student stuID="I7012-891" courseID="PSAT-080-5">
      <stu:lastName>Zheng</stu:lastName>
      <stu:firstName>Paddy</stu:firstName>
      <stu:examDate>2017-10-17</stu:examDate>
      <stu:pretest level="H">65</stu:pretest>
    </stu:student>

    <stu:student stuID="I8053-891" courseID="PSAT-080-5">
      <stu:lastName>Steinke</stu:lastName>
      <stu:firstName>Devon</stu:firstName>
      <stu:examDate>2017-10-22</stu:examDate>
      <stu:pretest level="M">51</stu:pretest>
    </stu:student>

    <stu:student stuID="I8154-741" courseID="PSAT-080-5">
      <stu:lastName>Browne</stu:lastName>
      <stu:firstName>Brenda</stu:firstName>
      <stu:examDate>2017-10-22</stu:examDate>
      <stu:pretest level="L">30</stu:pretest>
    </stu:student>
  </stu:students>
</crs:course>
```

- 4. Reload **psatstudents.xml** in your browser. Now that all the elements are qualified, the `lastName` and `firstName` elements for the instructor and the students are formatted using the rules for the course and students namespaces, respectively, as shown in Visual Overview 4.3.

**INSIGHT**

### Defining Namespaces with the Escape Character

Not all browsers support the use of the `@namespace` rule. When the specifications for XML 1.0 were first posted, no support existed for namespaces. Several competing proposals were circulated for adding namespace support to XML and CSS. One proposal, which was not adopted but was implemented in the Internet Explorer browser before version 9, was to insert the backslash escape character (\ ) before the colon character in the namespace prefix. So, for older versions of Internet Explorer to apply a style to an element from a particular namespace, you use the declaration

```
prefix\selector {attribute1:value1; attribute2:value2; ...}
```

where `prefix` is the namespace prefix used in the XML document. For example, the declaration for the `title` element in a products namespace that uses the `prd` prefix is as follows:

```
prd\title {width: 150px}
```

You can apply the same style to several elements in the namespace by using the \* symbol. For example, the following declaration sets the width of all elements in the products namespace to 150 pixels:

```
prd\*: {width: 150px}
```

Other browsers such as Firefox, Opera, and Safari do not support this method with XML documents. If you want to support the widest range of browsers, you must duplicate the styles in the style sheet using both methods.

Gabby is pleased that you were able to apply namespaces to the style sheets and the namespaces. The web page contains all the data that Gabby wants and is displayed in the way she intended. She'll base future documents for Higher Ed Test Prep on the model document you created.

### Session 4.3 Quick Check

1. What code would you add to an XML instance document to link it to the branding.css style sheet?
2. What rule would you add to a CSS style sheet to declare a namespace with the URI `http://ns.doc.student` and the namespace prefix `student`?
3. What rule would you add to a CSS style sheet to make the namespace in Question 2 the default namespace for all selectors in the style sheet?
4. In a style sheet that includes the namespace rule in Question 2, how would you modify the selector for the `lastname` element to indicate that it belongs to the namespace with the URI `http://ns.doc.student`?
5. What code would you add to a schema file to force all elements and all attributes to be qualified by default?

**PRACTICE**

## Review Assignments

**Data Files needed for the Review Assignments:** coursetxt.css, coursetxt.xsd, psattxt.xml, sessionstxt.css, sessionstxt.xml, sessionstxt.xsd

Gabby would like your help with creating another compound document and styling it with CSS. She would like to create a document that combines the description of a single course, using the course vocabulary, with a list of session descriptions for that course, using the sessions vocabulary. She then wants to apply the course.css and sessions.css styles to the content to produce formatted content in a web browser. Figure 4-29 shows a preview of the completed document.

Figure 4-29 Final PSAT Writing Skills Course compound document

The screenshot shows a web page titled "PSAT Writing Skills Course 3". At the top, there is a blue-bordered box containing descriptive text about the course. Below this, a numbered list of 8 items is displayed, each preceded by a red number (1 through 8) and a brief description. The text in the blue box and the list items are styled with a blue font color.

**PSAT Writing Skills Course 3**

This is a focused course for students who have previously taken the PSAT exam as a sophomore but did not receive a desired score on the writing skills portion or students who will be taking the PSAT and wish to strengthen their skills in the writing skills area. Outcomes of the course will be measured using the original PSAT writing skills score or the course pre-test compared to both the subsequent PSAT writing skills score and a course post-test.

- 1 How to identify sentence errors
- 2 Advanced identification of sentence errors
- 3 How to improve sentences
- 4 Advanced sentence improvement
- 5 How to improve paragraphs
- 6 Advanced paragraph improvement
- 7 Combining Skills I
- 8 Combining Skills II

Gabby has already created separate XML documents for the course and sessions content schemas to validate each vocabulary, and style sheets for the different elements in both vocabularies. She needs you to combine the content into a single document, import the sessions schema into the course schema, and edit the style sheets so that they support namespaces.

Complete the following:

1. In your XML editor, open the **psattxt.xml** and **sessionstxt.xml** documents and the **coursetxt.xsd** and **sessionstxt.xsd** schema files located in the **xml04 ▶ review** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **psat.xml**, **sessions.xml**, **course.xsd**, and **sessions.xsd**, respectively, in the same folder. Validate the psat.xml and sessions.xml files to confirm that they're valid.
2. In the sessions.xml file, copy the content from the opening **<sessions>** tag through the closing **</sessions>** tag, and then in psat.xml, paste the copied content directly before the closing **</course>** tag. Save the file as psatsessions.xml. Close the sessions.xml file.
3. In psatsessions.xml, in the opening **<course>** tag, keep the attribute and value that declare the XML Schema namespace, and then edit the attributes to declare the namespace <http://example.com/higheredtestprep/course/ns> for the course vocabulary, declare the namespace <http://example.com/higheredtestprep/sessions/ns> for the sessions vocabulary, and specify the location of the course.xsd schema file. Specify the prefix **crs** for the course namespace and the prefix **ses** for the sessions namespace. In the opening **<sessions>** tag, remove all attributes.

4. Qualify the `course` element and the `sessions` element using the prefixes declared in the previous step, and then save your work.
5. In the `course.xsd` file, specify the namespace  
`http://example.com/higheredtestprep/course/ns` for all unqualified names in the schema, and then specify the same namespace as the target namespace. Repeat for the `sessions.xsd` file, using the namespace `http://example.com/higheredtestprep/sessions/ns`.
6. In the `course.xsd` file, import the `sessions.xsd` file, specifying the `http://example.com/higheredtestprep/sessions/ns` namespace, declare the prefix `ses` for the sessions namespace, and then add a reference to the `sessions` element from the sessions namespace immediately after the declaration of the `description` element.
7. Save your work in all open files, and then validate `psatsessions.xml`.
8. In your XML editor, open the `coursetxt.css` and `sessionstxt.css` style sheets from the `xml04 ▶ review` folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as `course.css` and `sessions.css`, respectively, in the same folder.
9. In the `course.css` style sheet, declare  
`http://example.com/higheredtestprep/course/ns` as the default namespace. In the `sessions.css` style sheet, declare  
`http://example.com/higheredtestprep/sessions/ns` as the default namespace. Save your changes to both files.
10. In the `course.xsd` and `sessions.xsd` files, specify that elements are qualified by default, and that attributes are unqualified by default. Save your changes to both files.
11. In the `psatsessions.xml` file, add instructions to link to the `course.css` and `sessions.css` style sheets, and then qualify all elements. Save your work and validate the file.
12. Open `psatsessions.xml` in your browser and verify that it matches Figure 4-29.

**APPLY**

## Case Problem 1

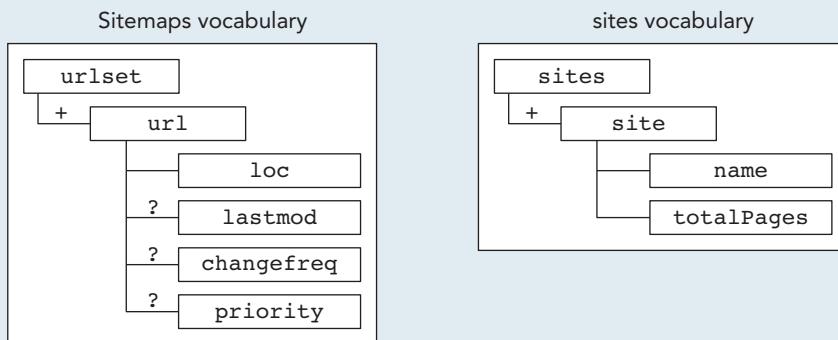
**Data Files needed for this Case Problem:** `sitemapsPS.xml`, `sitemapsVS.xml`, `sitemapsWFS.xml`, `sitetxt.xml`, `sitetxt.xsd`

**Weekend Fun Snacks** Cleo Coal created and maintains a website called Weekend Fun Snacks, which lists her picks of the best and easiest recipes for kids to cook. The site's popularity convinced her there was room for more specialty recipe sites, so she created two additional websites—Primal Snacks, which features snacks appropriate for a paleo or primal diet, and Veg Snacks, which includes quick bites suitable for vegetarians.

Cleo also created a Sitemaps file for each site, which is a document written in XML that provides basic information about each page in a website, as well as how all the pages are related. Cleo has submitted her Sitemaps files to Google and other search services to help them better index her sites. However, she also thinks that the Sitemaps content she's created would be useful in a compound document with a custom vocabulary, which would allow her to view information about all the pages on all of her sites in a single page. Cleo asks for your help with creating this compound document.

Figure 4-30 shows a tree diagram highlighting some of the elements from both vocabularies that you'll place in the document.

**Figure 4-30 Tree diagrams of Sitemaps and sites vocabularies**



Cleo has provided you with a truncated version of the Sitemaps file for each of her three websites as well as another XML file containing the administrative information on her sites. Your job will be to create a compound document combining the features of the two vocabularies.

Complete the following:

1. In your XML editor, open the **sitestxt.xml** and **sitestxt.xsd** files from the `xml04 ▶ case1` folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **sites.xml** and **sites.xsd**, respectively, in the same folder.
2. In the `sites.xml` file, add a namespace declaration to the root `sites` element, associating the `xs` prefix with the URI for the XML Schema namespace. Specify the default namespace `http://example.com/weekendfunsnacks/sites` for the file. Specify `sites.xsd` as the location of the schema for the default namespace.
3. In your XML editor, open the **sitemapPS.xml**, **sitemapVS.xml**, and **sitemapWFS.xml** files from the `xml04 ▶ case1` folder. These files contain the Sitemaps for Cleo's three websites. In the `sitemapWFS.xml` file, copy the contents from the opening `<urlset>` tag through the closing `</urlset>` tag to the Clipboard, and then paste them into the `sites.xml` file just before the closing `</site>` tag for the Weekend Fun Snacks site. Repeat to copy and paste the content from the `sitemapPS.xml` file into the `site` element for the Paleo Snacks site, and the content from the `sitemapVS.xml` file into the `site` element for the Veg Snacks site. Save your changes to the `sites.xml` file.
4. In each of the opening `<urlset>` tags you pasted in the previous step, remove the XML Schema namespace declaration and the schema location, leaving the default namespace declaration for each element. Save your work.
5. In the `sites.xsd` file, in the root element, specify the target namespace as `http://example.com/weekendfunsnacks/sites`, and then associate the prefix `cc` with the target namespace. Associate the prefix `sm` with the namespace `http://www.sitemaps.org/schemas/sitemap/0.9`. Specify that elements are qualified by default, and that attributes are unqualified by default.
6. Add code to import the schema for the  
`http://www.sitemaps.org/schemas/sitemap/0.9`  
namespace from the location  
`http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd`.
7. Immediately following the declaration of the `totalPages` element, add a reference to the `urlset` element from the  
`http://www.sitemaps.org/schemas/sitemap/0.9` namespace.  
Save your work.
8. Validate the `sites.xml` file, and then, if necessary, fix any validation errors.

## Case Problem 2

**Data Files needed for this Case Problem:** `menutxt.css`, `menutxt.xml`, `recipetxt.css`, `recipetxt.xml`

**Chester's Restaurant** Chester's Restaurant is located in Hartland, Minnesota. Jasmine Pup, the owner and operator, has all the menu information stored in XML files. She also has the recipes for all the menu items stored in XML. She'd like to combine the menu and recipe information for each item into a document formatted with CSS, which would give her an easy-to-read overview of the description and ingredients of each menu item.

Jasmine has asked you to combine the menu and recipe information about one menu item for her approval. Figure 4-31 shows tree diagrams of the subsets of the menu and recipe vocabularies that you'll be using.

Figure 4-31 Tree diagrams of menu and recipe subsets

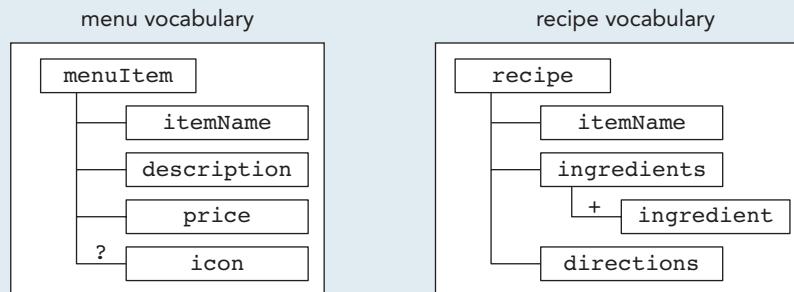


Figure 4-32 shows a preview of the page that you will create.

Figure 4-32 Preview of menu and recipe document in browser

The screenshot shows a web browser displaying a menu item for 'Oatmeal Breakfast'. The item is described as 'Our oatmeal is served warm with fresh fruit, pecans, raisins, and 100% maple syrup. Available all day.' with a price of '6.95' and icons for a fork and knife. Below this, a box contains the recipe ingredients: '1/3 c steel cut oats', '1-1/4 c water', and '1/4 t salt'. A note at the bottom says: 'Bring water to a boil. Add salt and oats, stir, and lower heat to lowest setting. Cover and let stand 2 hours.'

Complete the following:

1. In your XML editor, open `menutxt.xml` and `recipetxt.xml` from the `xml04 ▶ case2` folder, enter **your name** and **today's date** in the comment section of each file, and then save the documents as `menu.xml` and `recipe.xml`, respectively, in the same folder.
2. Review the contents of `menu.xml`, and then create a schema file for the menu vocabulary using the Russian Doll design. Save the schema file to the `xml04 ▶ case2` folder as `menu.xsd`. Specify the target namespace for the schema and save your changes. In `menu.xml`, specify the location of the schema file and save your changes. Validate `menu.xml` against the schema and then, if necessary, fix any validation errors.

3. Review the contents of recipe.xml, and then create a schema file for the recipe vocabulary using the Russian Doll design. Save the schema file to the xml04 ▶ case2 folder as **recipe.xsd**. Specify the target namespace for the schema and save your changes. In recipe.xml, specify the location of the schema file and save your changes. Validate recipe.xml against the schema and then, if necessary, fix any validation errors.
4. In the recipe.xml file, copy the `recipe` element and its contents to the Clipboard, and then in the menu.xml file, paste the recipe contents directly before the closing `</menuItem>` tag. Save a copy of the file as **menurecipe.xml**, and then change the filename listed in the comment section to match.
5. In menurecipe.xml, select and assign a prefix to the XML Schema namespace. Select and assign a prefix to the menu namespace specified in the root element of the menu.xml file.
6. Select and assign a prefix to the recipe namespace specified in the root element of the menurecipe.xml file. Specify the location of the schema file for the menu namespace.
7. Qualify all elements in the menurecipe.xml file using the prefixes you defined.
8. In menu.xsd, specify that elements are qualified by default and attributes are unqualified by default. Specify the namespace of the menu vocabulary as the target namespace. Assign a prefix to the namespace for the recipe vocabulary. Import the recipe.xsd file, and then add a reference to the `recipe` element from the recipe namespace immediately after the definition of the `icon` element. Save your work.
9. In your XML editor, open **menutxt.css** and **recipetxt.css** from the xml04 ▶ case2 folder, enter **your name** and **today's date** in the comment section of each file, and then save the documents as **menu.css** and **recipe.css**, respectively, in the same folder.
10. In menu.css, specify the namespace for the menu vocabulary as the default namespace, and then save your changes.
11. In recipe.css, specify the namespace for the recipe vocabulary as the default namespace, and then save your changes.
12. In menurecipe.xml, directly below the comment section, insert a processing instruction that links the document to the **menu.css** style sheet. Insert another processing instruction that links the document to the **recipe.css** style sheet. Save your changes.
13. Validate the menurecipe.xml file and then, if necessary, fix any validation errors.
14. Open menurecipe.xml in your browser and verify that its appearance matches Figure 4-32.

**CHALLENGE**

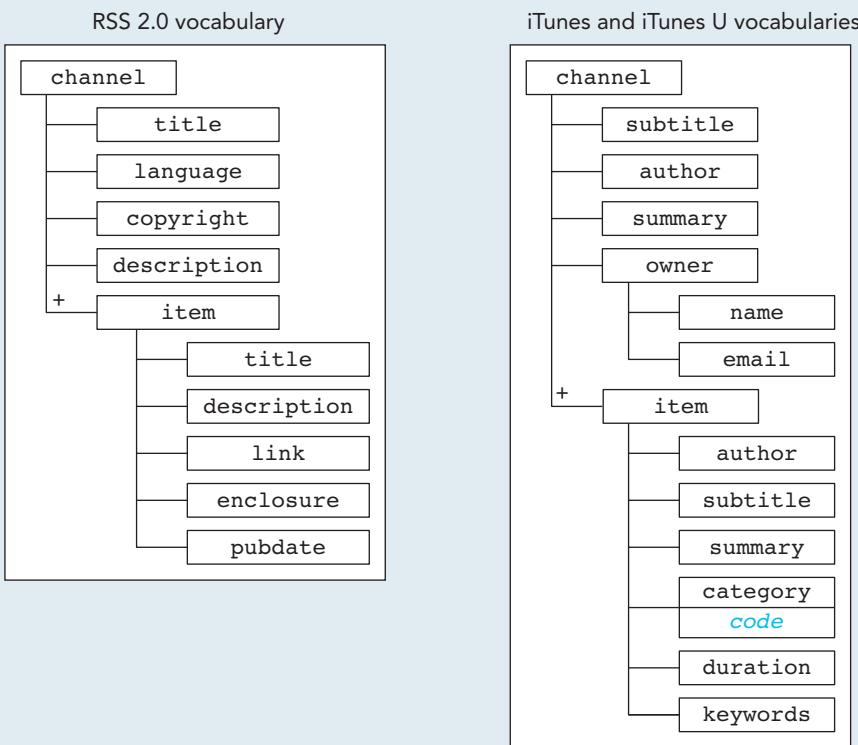
## Case Problem 3

**Data Files needed for this Case Problem:** atclectxt.xml, ituneselem.txt

**Austin Technical College** Pia Zhou is a community relations officer for the School of Information Technology at Austin Technical College (ATC) in Austin, Utah. Pia is part of an interdepartmental task force exploring ways to make more ATC courses available online. Her current task is to explore making lectures from a single class available through iTunes U, which is an Apple application that lets instructors distribute course content to students. Pia has asked for your help with creating an initial demonstration document.

iTunes U content is formatted in XML using the RSS vocabulary, supplemented with elements from the custom iTunes and iTunes U vocabularies.

Figure 4-33 shows tree diagrams highlighting some of the elements from the vocabularies that you'll place in the document.

**Figure 4-33** Tree diagrams of RSS and iTunes vocabularies

You've already received an RSS file containing the RSS elements and a text file containing iTunes U-related information on the course and lectures. Your job will be to create a compound document combining the features of the two vocabularies. Note that by convention, RSS documents do not declare the RSS namespace.

Complete the following:

1. In your XML editor, open the **atclectxt.xml** file from the **xml04 ▶ case3** folder, enter **your name** and **today's date** in the comment section, and then save the file as **atlecture.xml** in the same folder.
2. Add a namespace declaration to the root **rss** element declaring the iTunes namespace <http://www.itunes.com/dtds/podcast-1.0.dtd>. Use **itunes** as the namespace prefix. Add a second namespace declaration to the root **rss** element declaring the iTunes U namespace <http://www.itunesu.com/feed>. Use **itunesu** as the namespace prefix.
- EXPLORE** 3. In your XML editor, open the **ituneselem.txt** file from the **xml04 ▶ case3** folder. This file contains the content for the different iTunes and iTunes U elements. Using this file as a reference, complete the rest of the content in the RSS document.
4. Return to the **atlecture.xml** file in your XML editor. Add the **subtitle**, **author**, and **summary** iTunes elements as child elements of the **channel** element. Place all three elements in the iTunes namespace, and use the text indicated in the **ituneselem.txt** file as the content of the three elements.
5. Below the **description** element, insert the iTunes **owner** element. The **owner** element indicates the owner of the podcast for the iTunes Store. Within the **owner** element, insert two elements named **name** and **email** containing Zakia Choudhry's name and email address, respectively. Make sure these elements belong to the iTunes namespace.
6. Add iTunes elements that describe each lecture in the series. Each lecture is marked with the **item** element. Zakia's document includes four lectures. Add the **author**, **subtitle**, **summary**, **duration**, and **keywords** elements for each of the four lectures.

7. Specify the category of each lecture using the iTunes U `category` element. The name of the category is contained in an attribute of the `category` element named `code`. The `code` attribute for each lecture should have a value of 101102, which corresponds to computer science.
8. Save your changes to the file, and then load `atlecture.xml` in your web browser. Verify that no errors are reported in the document.

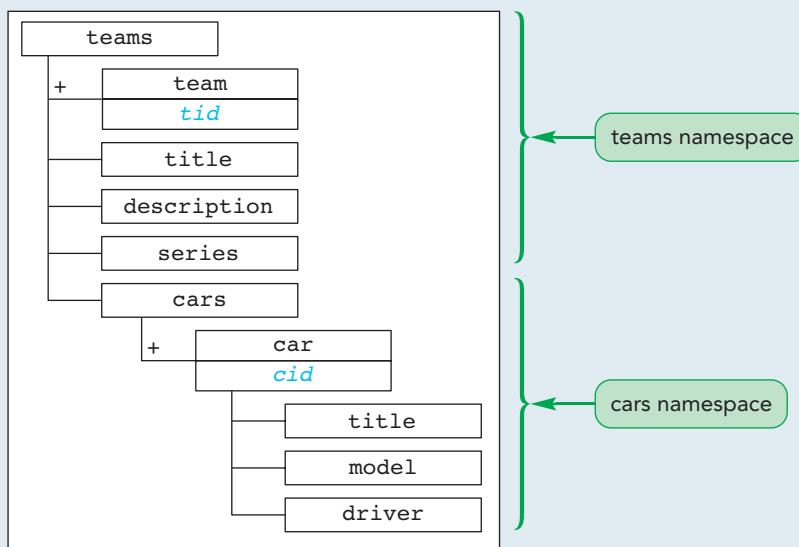
CREATE

## Case Problem 4

**Data Files needed for this Case Problem:** `carstxt.css`, `carstxt.xml`, `teamstxt.css`, `teamstxt.xml`

**South Racing** Danika Francis tracks team cars for South Racing's racing teams. As part of her job, she has created several XML vocabularies dealing with team series and the cars available to race in them. She has created an XML document containing information on two teams, and another document on cars. She wants to combine this information to create a compound document that includes team information and lists each team's cars. She has already developed a style sheet for each vocabulary. Figure 4-34 shows the tree structures of the teams and cars vocabularies.

Figure 4-34 Tree diagrams of combined teams and cars vocabularies



Danika has asked you to create schema documents to validate both vocabularies, and a compound document that incorporates the information and style sheet formatting from both vocabularies. There is some overlap in the element names from the two vocabularies, so you'll have to use namespaces to distinguish the elements from the two vocabularies. Figure 4-35 shows a preview of the page you'll create.

Figure 4-35 Completed South Racing document in browser

<b>Rodas Motorsports</b>	
<b>#1 Team in racing</b>	Indy
• Straight Away	Nissan indy
• Quick Start	General Motors indy
• Stop Blocks Laps	Ford indy
• Bendwinder	Ford 412
• Turn Twister	Nissan indy

<b>SAM Racing</b>	
<b>Top 10 over last 5 years</b>	Nascar
• Straight Away	Nissan indy
• Quick Start	General Motors indy
• Stop Blocks Laps	Ford indy
• Sleeker	General Motors 278
• 84 Racer	General Motors 198

Complete the following:

1. In your XML editor, open the **carstxt.css**, **carstxt.xml**, **teamstxt.css**, and **teamstxt.xml** files from the **xml04 ▶ case4** folder, enter **your name** and **today's date** in the comment section of each file, and then save the files as **cars.css**, **cars.xml**, **teams.css**, and **teams.xml**, respectively, in the same folder.
2. Create a schema file for each of the two XML files using whichever schema design you choose, and selecting appropriate filenames and namespaces. Validate each XML file against its schema file, and then correct any errors, if necessary, until both instance documents validate.
3. Add a **cars** element after the **series** element for both of the teams. Copy and paste the relevant **cars** elements, along with their children, for each team, as shown in Figure 4-34. Save the compound document as **teamscars.xml**.
4. Associate the schema information with the compound document.
5. Specify the default namespace for each style sheet, and then link the compound document to both style sheets.
6. Validate the compound document and then correct any errors, if necessary, until it validates. Open the document in your browser and verify that it matches Figure 4-35.



# Decision Making

## Deciding How to Structure Data with XML

Decision making is a process of choosing between alternative courses of action. The steps involved in evaluating a given alternative include the following:

1. Obtain relevant information.
2. Make predictions about the future.
3. Select the best alternative.
4. Prepare an action plan to implement the alternative.
5. Launch the implementation and monitor the result.
6. Verify the accuracy of the decision and take corrective action, if needed.

For some decisions, you might combine some steps, and you might even skip steps for the simplest decisions.

### Obtaining Information, Making Predictions, and Selecting the Best Alternative

In order to effectively evaluate a potential course of action, data and information must be gathered. The relevant information may include quantitative financial factors that can be expressed in monetary or numerical terms, and qualitative factors that cannot be measured in numerical terms. For example, a company thinking about switching to a new system for maintaining electronic data will gather quantitative information related to the costs of the current system, such as staff time, salaried positions, software licensing, and hardware requirements. Additional information may include qualitative information related to factors such as employee morale.

After collecting relevant information, a decision model can be used to help make predictions about how the costs, behaviors, and states of nature beyond the control of the decision maker may influence outcomes. Excel spreadsheets are well suited to the quantitative portion of this task; qualitative variables may be assigned numerical weights so they, too, can be part of the decision model used for predicting potential outcomes.

Using quantitative approaches to making a decision can lead to greater confidence in the choice, but you should not ignore the value of qualitative information. After modeling the decision alternatives and calculating outcomes, selection of the best alternative may require asking additional questions, such as:

- What qualitative factors must be considered in addition to the quantitative analysis, and do they carry enough weight to discount one or more options?
- Does this alternative make sense for the long term?
- Can this alternative be realistically implemented? Think about resources and time frame, for example.
- Will the alternative be acceptable even if the outcome is not perfect, or if some unconsidered factors emerge after implementation?

### Preparing an Implementation Action Plan

Once the decision has been made, the steps necessary to implement the decision must be determined. The decision maker should have a pretty good idea of what the final outcome should be in order to consider all relevant steps. For example, in the case of moving to a new data storage system, the final outcome is new software and hardware in use company-wide.

One key consideration is the time table for implementation. When will it start? How long will each task take? What tasks must be completed before others start? Can tasks be performed concurrently?

A project manager also must be chosen to help develop and manage the implementation action plan. This person will be held accountable for all tasks, resources, and scheduling to assure the decision is implemented as originally designed.

Key milestones for the implementation must be determined so that successful completion can be tracked. Determining who will be accountable for these milestones can help keep track of completion. The project manager may have overall responsibility for keeping the implementation on budget and on time, but others will play a supporting role in getting work done.

What resources are required for successful implementation? Money? Personnel? Facilities? Are these available in-house, or does external expertise need to be sourced?

Often, the most challenging part of implementing some decisions is dealing with the human and behavioral aspects. Part of the action plan must consider regular communication with all affected parties, including weekly project status updates, scheduled training sessions, and mechanisms for handling inquiries, feedback, or opposition.

## Taking Action and Monitoring Results

Once the decision is made, approvals are received, and the action plan is developed, the actual implementation of the plan can begin. As progress is made, completion of the predetermined tasks can be documented and assessed against the schedule. The project manager can then compare actual completion activity against planned activity to be sure the implementation stays on track.

Occasionally, the best-laid plans do veer off-course. In this case, the project manager must be able to determine why, when, and where the tasks fell behind schedule to help set them back on course.

## Verifying the Accuracy of the Decision

Once the action plan has been implemented, it is essential to verify that the decision was the correct course of action. The decision maker can assess the effect of the implemented decision by collecting feedback about the changes in operations. For example, in the case of the new electronic data storage system, were the anticipated cost savings achieved? Was the retraining of affected staff managed appropriately?

### Design and Implement a Custom XML Vocabulary

XML is a powerful tool for developing structured documents whose content can be tested against a collection of rules defined in a DTD or schema. In this exercise, you'll use XML to create a vocabulary, and then you'll create and validate an XML document using your vocabulary by applying the XML skills you've learned in these tutorials.

**Note:** Please be sure *not* to include any personal information of a sensitive nature in the documents you create to be submitted to your instructor for this exercise. Later on, you can update the data in your database with such information for your own personal use.

1. Design your own XML vocabulary for a field of study that interests you. Your vocabulary should include the following features:
  - a. Elements containing textual content
  - b. Elements containing child elements
  - c. Attributes containing textual content
  - d. XML-supported entities
2. Write a summary documenting your vocabulary for other users.

3. Create an instance document based on your XML vocabulary.
4. Write a DTD to validate your instance document based on your vocabulary. Confirm that your instance document passes validation.
5. Write a schema to validate your instance document. Your schema should include the following features:
  - a. One or more custom data types
  - b. A named complex type
  - c. Schema contents laid out in a Venetian Blind design
6. Apply your schema to your instance document and verify that your instance document passes validation.
7. Create a second XML vocabulary in the same field of study as your first. Document your vocabulary for other users.
8. Create a namespace for each of your vocabularies.
9. Create a compound document combining elements and attributes from both of your vocabularies.
10. Create a second schema file to validate the contents of your second XML vocabulary.
11. Apply your combined schemas to your compound document and confirm that the compound document passes validation.
12. Document your code and describe what you've learned from creating your own system of XML documents.

**OBJECTIVES****Session 5.1**

- Learn the history and theory of XSLT
- Understand XPath and examine a node tree
- Create and attach an XSLT style sheet
- Create a root template
- Generate a result document from an XSLT style sheet

**Session 5.2**

- Create and apply templates to different nodes
- Extract and display the value of an element
- Extract and display the value of an attribute
- Explore XSLT's built-in templates

**Session 5.3**

- Set the value of an attribute in a result document
- Create conditional output using the `if` and `choose` elements
- Create an XPath expression using predicates
- Use XSLT to generate elements and attributes

**STARTING DATA FILES**

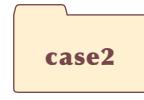
stocktxt.xml  
stocktxt.xsl  
+ 1 CSS file  
+ 18 PNG files



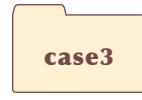
portfoliotxt.xml  
portfoliotxt.xsl  
+ 1 CSS file  
+ 3 PNG files



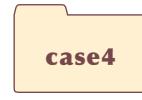
itemstxt.xml  
librarytxt.xsl  
+ 1 CSS file  
+ 2 PNG files



feedtxt.xsl  
newstxt.xml  
+ 1 CSS file  
+ 5 HTML files

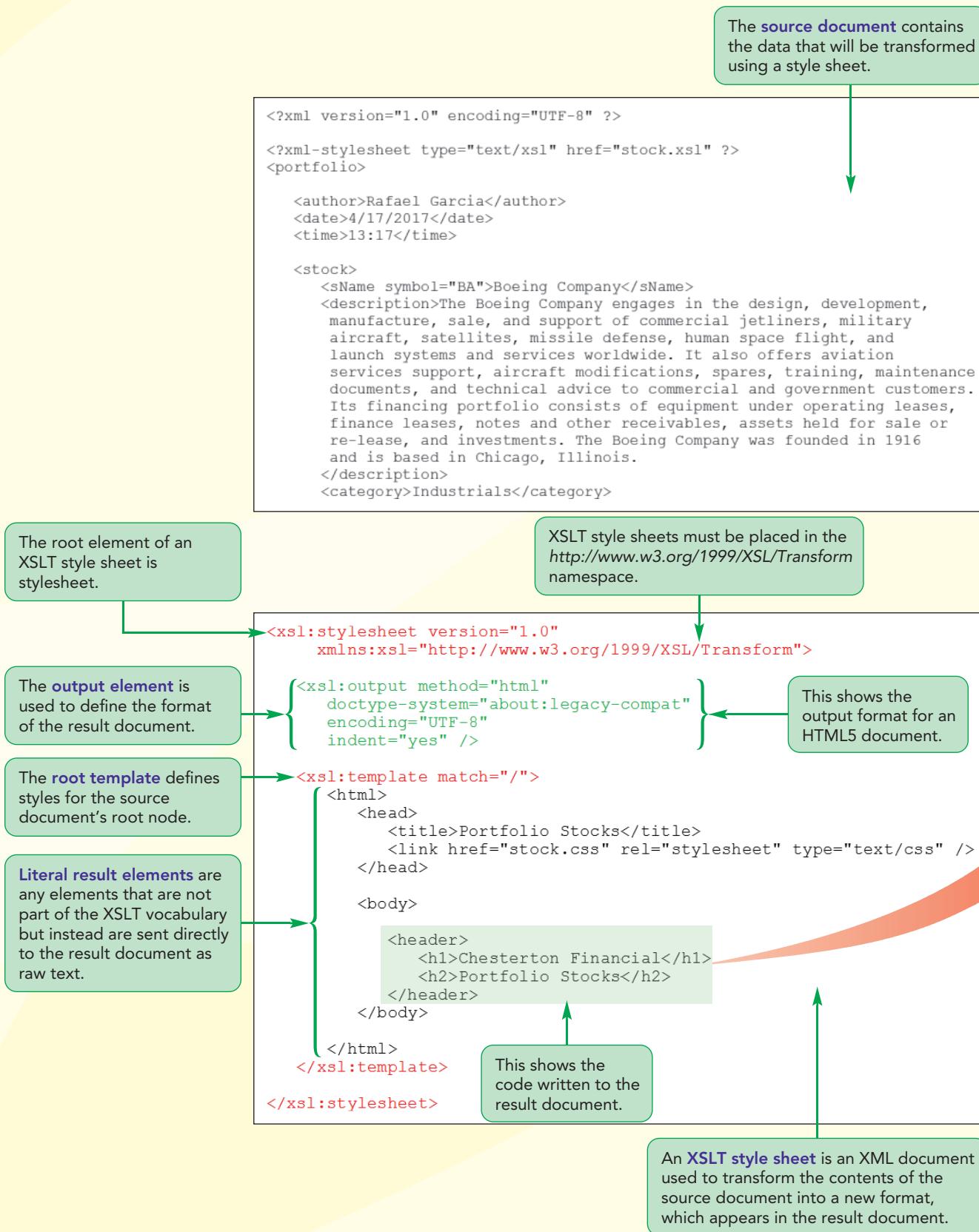


clisttxt.xsl  
orderstxt.xml  
+ 1 DTD file

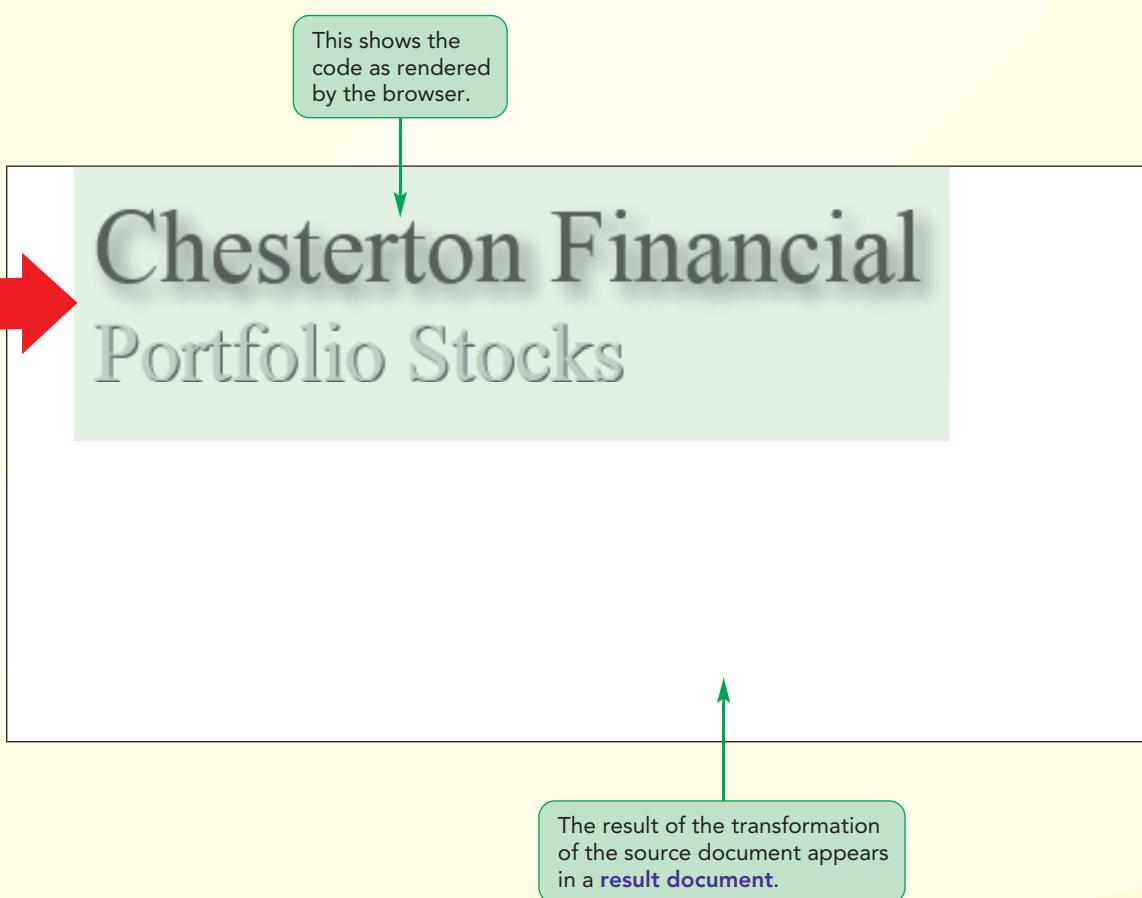


campingtxt.xml  
campingtxt.xsl

# Session 5.1 Visual Overview:



# Creating an XSLT Style Sheet



## Introducing XSL and XSLT

A challenge of working with data stored in XML is presenting that data in an easily readable format. One way of achieving this is by using [Extensible Stylesheet Language](#) or [XSL](#). XSL is used to transform the contents of a source XML document containing data into a result document written in a new format. XSL is itself an XML vocabulary, so you can apply much of what you've learned in writing XML code toward writing your first XSL style sheet. XSL is organized into two languages:

- [XSL-FO \(Extensible Stylesheet Language – Formatting Objects\)](#) is used for the layout of paginated documents.
- [XSLT \(Extensible Stylesheet Language Transformations\)](#) is used to transform the contents of an XML document into another document format.

XSL-FO describes the precise layout of text on a page, indicating the placement of individual pages, text blocks, horizontal rules, headers, footers, and other page elements. XSLT is used for text-based output such as HTML and XHTML (for creating web pages), Portable Document Format (PDF), Rich Text Format (RTF), and so forth. XSLT can also take an XML file and rewrite it as a new XML document with a different structure and set of elements and attributes. In this tutorial you'll work only with XSLT, transforming a source document into HTML code that can then be displayed on a website.

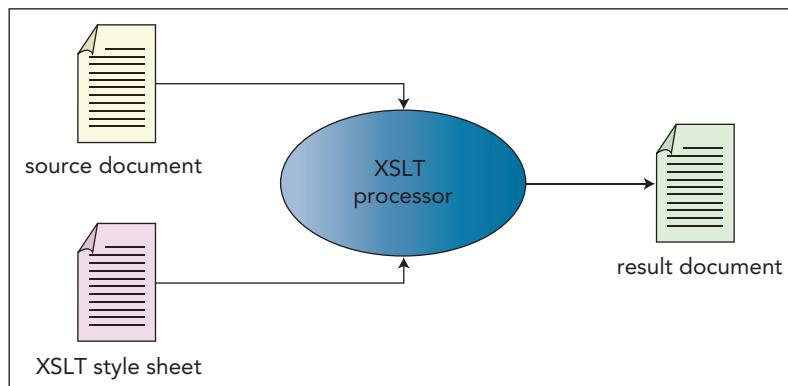
There are several versions of XSLT. XSLT 1.0 is the original specification finalized by the W3C in 1999. The follow-up version, XSLT 2.0, reached recommendation status in 2007. Support for XSLT 2.0 was slow to develop; thus, it is not unusual at this time to find legacy applications based solely on XSLT 1.0 standards. XSLT 3.0 extends the language further by supporting data streaming, which allows the output document to be processed as the input document is read, resulting in greater speed and efficiency. Currently XSLT 3.0 is in the draft stage of development.

## XSLT Style Sheets and Processors

Because XSLT is itself an XML vocabulary, you can create an XSLT style sheet using a basic text editor. XSLT style sheets have the filename extension .xsl to distinguish them from other XML documents. Once your style sheet is written, an XSLT processor is used to transform the contents of the source document into a new format (see Figure 5-1), which appears as the result document.

**Figure 5-1**

Transforming a source document



The transformation can be performed on a server or a client. In a [server-side transformation](#), a server receives a request from a client to generate the result document. The server applies the style sheet to the source document and returns the result document to the client, often as a new file. In a server-side transformation, the client does not need an XSLT processor because all of the work is done on the server. This makes the process

more accessible to a wide variety of users who may not have access to an XSLT processor. A disadvantage to server-side transformations is the heavy load they can place on a server as it attempts to handle transformation requests from multiple clients.

In a **client-side transformation**, a client requests retrieval of both a source document and a style sheet from the server. The client then performs the transformation and generates its own result document. There are several client-side XSLT processors available, including the following:

- Altova Raptor/XML Server is a processor sold with XMLSpy and supports XSLT 1.0, 2.0, and 3.0.
- libxslt is a free library of functions written in C supporting XSLT 1.0 and used within WebKit to run transformations within the Safari and Google Chrome browsers.
- MSXML is a set of application services supporting XSLT 1.0 that was developed by Microsoft and is included with Internet Explorer.
- Saxon is an open source standalone processor supporting XSLT 2.0 that can be used with Java and JavaScript applications.
- Xalan Apache is an open-source component that supports XSLT 1.0 and was developed by the Apache Software Foundation for both the Java and C++ programming languages.

#### TIP

For security reasons, Google Chrome displays transformations only for files stored on a server but not for files saved locally.

Most current browsers have built-in XSLT processors supporting XSLT 1.0. Thus, to view the results of a transformation, you only need to open the source XML document within your browser. If you want the transformed document stored as a separate file or if you want to use XSLT 2.0, you will need an XML editor. There are several free and commercial editors that support XSLT 1.0 and 2.0; the most prominent at the time of this writing is the free home edition of the Saxon processor.

## Attaching an XSLT Style Sheet

An XSLT style sheet is attached to an XML document by adding the following processing instruction near the top of the XML document prior to the root element:

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

where *url* is the URL pointing to the location of the XSLT style sheet file.

Now that you have learned the basics about XSLT, you'll begin helping Rafael to develop an XSLT style sheet for displaying his stock market data. Rafael has a source XML document containing data from 15 different stocks in a sample portfolio. He wants to display this information in a web page, generating the HTML code with his XSLT style sheet. You'll start developing the XSLT style sheet by adding a processing instruction to Rafael's source document, linking that file to a proposed style sheet file.

### To add the processing instruction:

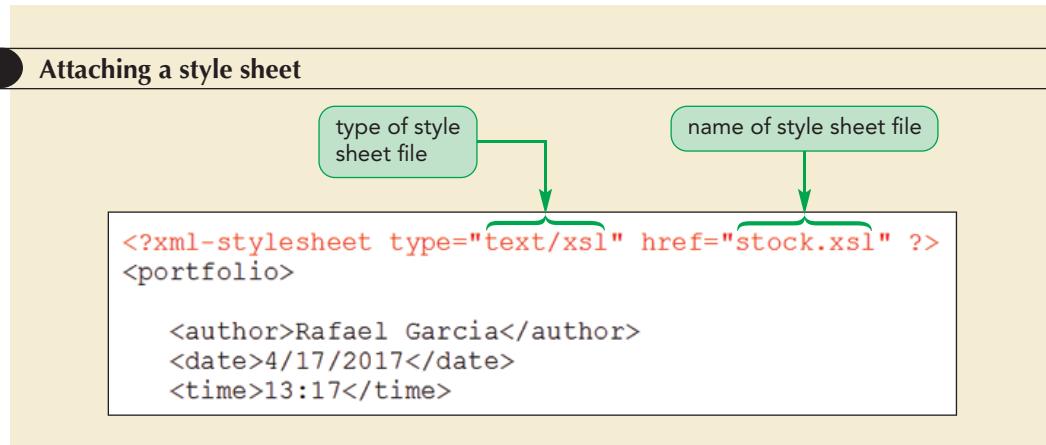
1. Use your text editor to open **stocktxt.xml** from the **xml05 ▶ tutorial** folder. Enter **your name** and the **date** in the comment section at the top of the file, and save the file as **stock.xml**.
2. Insert the following processing instruction directly above the opening **<portfolio>** tag at the top of the file:  

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

Figure 5-2 shows the newly added processing instruction.

Figure 5-2

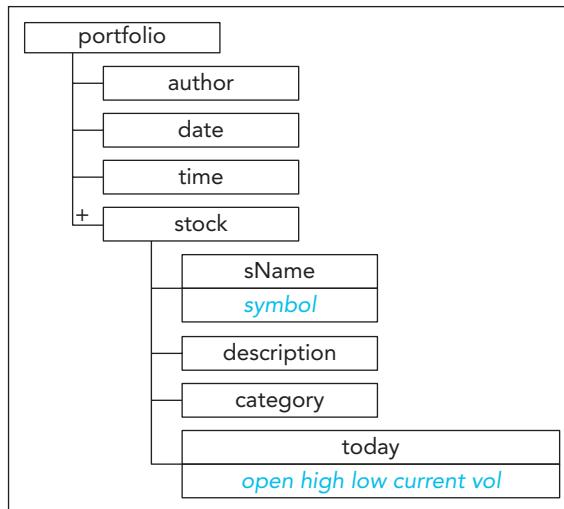
Attaching a style sheet



Next, you and Rafael discuss how he wants to have the data in the stock.xml file transformed into an HTML document. First you need to study the current content and layout of the source file. Figure 5-3 shows the document's structure.

Figure 5-3

Structure of the stock.xml file



The content and purpose of each element and attribute is described in Figure 5-4.

Figure 5-4

Contents of the stock.xml file

Element	Description
portfolio	The root element
author	The author of the document
date	The date of the document contents
time	The time the document was last updated, in 24 hour format
stock	Information for an individual stock
sName	The name of the stock, containing an attribute named symbol that stores the stock ticker symbol
description	A description of the stock
category	The category for the stock: Industrials, Transportation, or Utilities
today	The opening, high, low, current, and volume values of the stock, stored in the following attributes: open, high, low, current, and vol

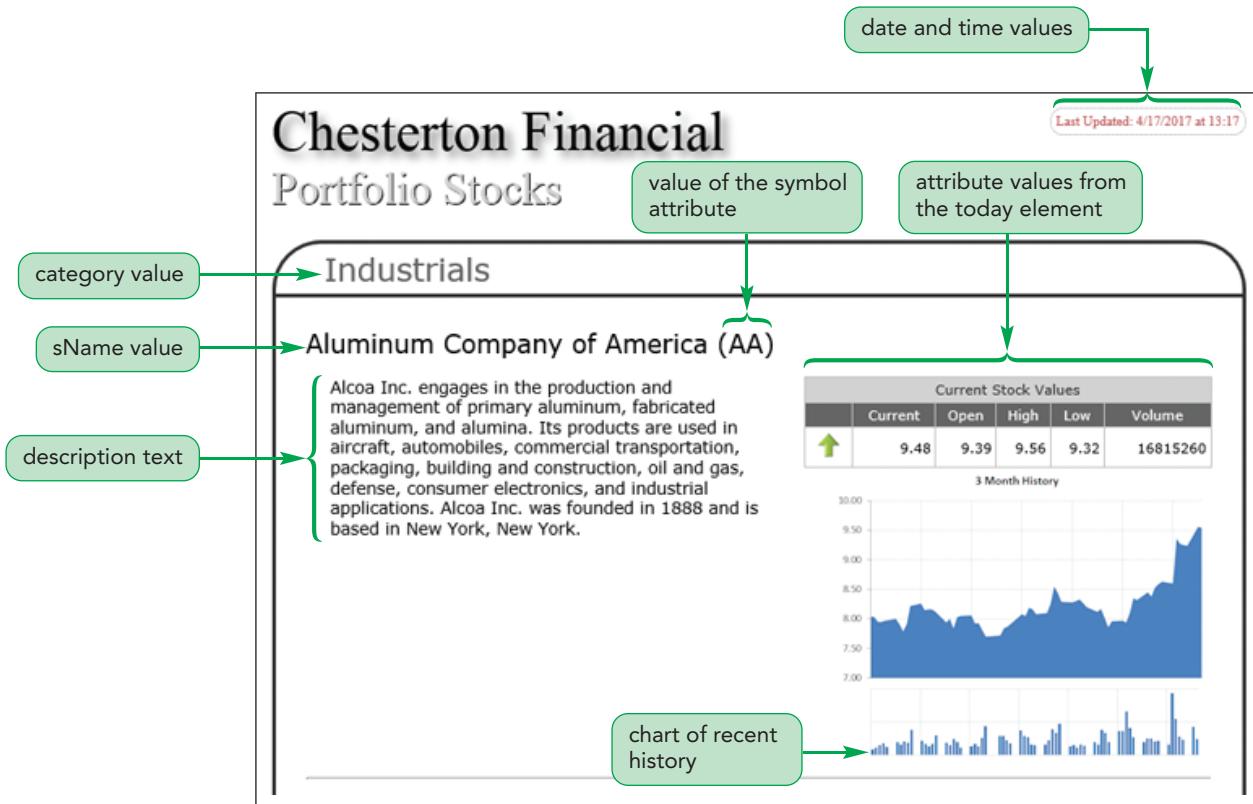
Take a few minutes now to review the contents of the file, comparing the structure and content to Figures 5-3 and 5-4.

### To review the stock.xml file:

- 1. Scroll down through the stock.xml file using your editor, paying attention to the use of elements and attributes throughout the document.
- 2. After you have finished reviewing the document, close the file, saving your changes.

Figure 5-5 previews how Rafael wants the data in the stock.xml to be transformed into a web page. The date and time values are placed at the top-right corner of the page so that readers can quickly determine how current the data is. The stock's name, description, and ticker symbol are drawn from the sName and description elements, and the symbol attribute. Attributes from the today element are used to populate the table of current stock values. An inline image displaying a chart of the last three months of stock activity has been added to the web page. Finally, note that Rafael wants the stocks sorted into different stock categories (Industrials, Transportation, or Utilities) so stocks that share common characteristics appear together.

**Figure 5-5** Preview of the transformed stock data



Now that you've seen how Rafael wants the output document to appear, you can begin writing the XSLT style sheet file to generate the HTML code.

## Starting an XSLT Style Sheet

Because XSLT style sheets are XML documents, XSLT documents start with an xml declaration and a root element named `stylesheet`. The `stylesheet` element needs to be placed in the `http://www.w3.org/1999/XSL/Transform` namespace. Thus, every XSLT stylesheet has the following basic structure:

### TIP

Some older XSLT style sheets might use the tag `<xsl:transform>` in place of `<xsl:stylesheet>`. Both are supported, but the `<xsl:stylesheet>` tag is preferred.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="value"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    style sheet contents
</xsl:stylesheet>
```

where `value` is the XSLT version and `style sheet contents` are the elements and attributes specific to the style sheet.

For Rafael's project you'll use a version number of "1.0" to indicate to the XSLT processor that this file should be compliant with XSLT 1.0 standards. You create the `stock.xsl` style sheet file now.

### To create the XSLT style sheet:

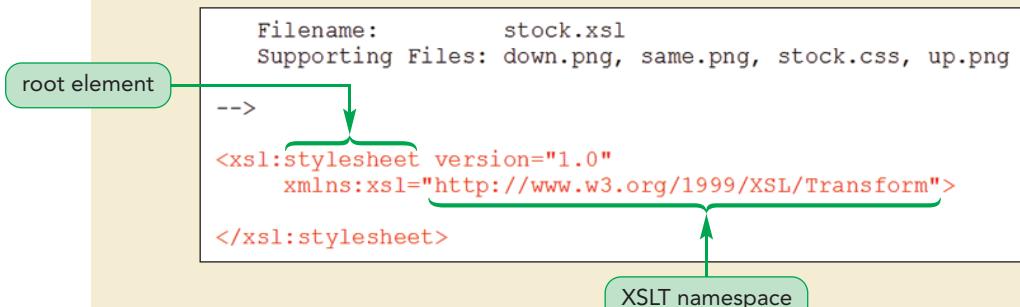
- 1. Use your text editor to open `stocktxt.xsl` from the `xml05 ▶ tutorial` folder. Enter **your name** and the **date** in the comment section at the top of the file, and save the file as `stock.xsl`.
- 2. Below the comment section insert the following tags:

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

Figure 5-6 highlights the root `stylesheet` element used in the document.

Figure 5-6

Root element of an XSLT style sheet



- 3. Save your changes to the `stock.xsl` file.

With the initial structure of the XSLT file in place and linked to the XML source document, `stock.xml`, you can begin working with the style sheet design. To do this, you first need to learn how to access the source document content from within the style sheet.

## Introducing XPath

While XML data is stored in text files, the contents are read into memory and stored in a hierachal tree structure. The **XPath** language, which was introduced by W3C, is used to access and navigate the contents of that data tree. Like XSLT, XPath has gone through several versions, usually paired with enhancements of XSLT. The initial version, XPath 1.0, was released in 1999 and enjoys universal support with XML processors and web browsers. XPath 2.0 reached Recommendation status in 2007 and is the most current version of the language. Processors that support XSLT 2.0 will also support most, if not all, features of XPath 2.0. None of the major web browsers provides built-in support for XPath 2.0 at the time of this writing. XPath 3.0, like XSLT 3.0, is only in the candidate stage of language development and is not widely supported by XML processors or web browsers.

## Working with Nodes

XPath operates by expressing the contents of the source document in terms of nodes. A **node** is any item within the tree structure of the document. A collection of nodes is called a **node set**. Any of the following objects from an XML document are considered nodes of that document:

- An element
- The text contained within an element
- An element attribute
- A comment statement
- A processing instruction
- A defined namespace
- The entire source document itself

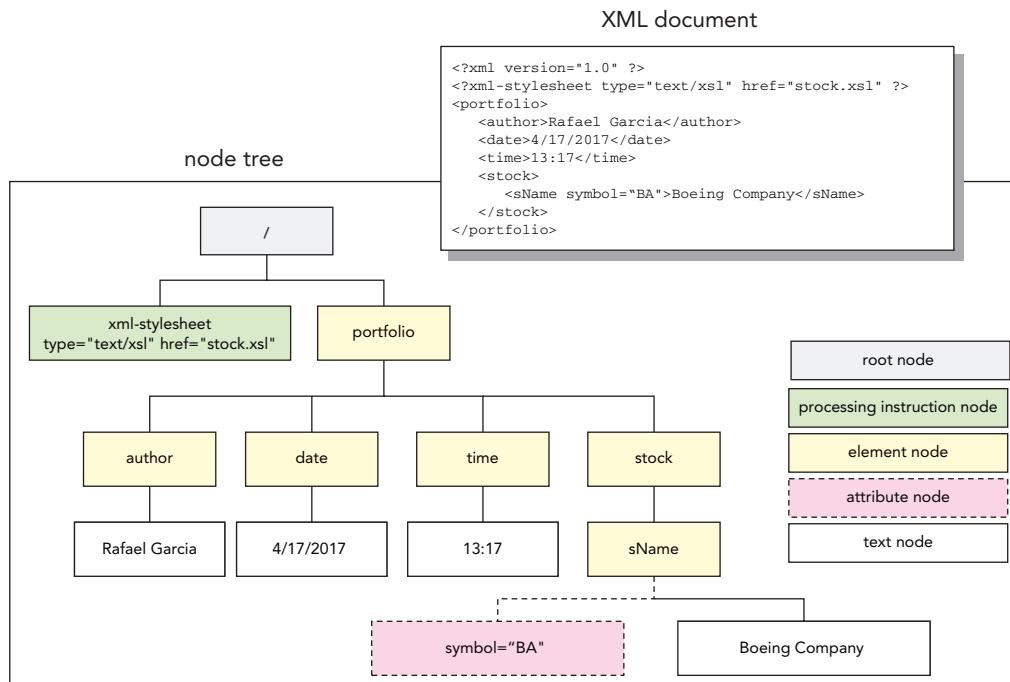
Nodes are referred to based on the type of objects they contain. Thus, an **element node** refers to an element from the source document, an **attribute node** refers to an element's attribute, and so forth.

However, not everything within the source document is a node. The following are *not* considered nodes:

- The XML declaration
- A DOCTYPE declaration
- A CDATA section
- An entity reference

The various nodes from the source document are organized into a **node tree**, with the **root node** or **document node** at the top of the tree. The root node contains all other nodes. It is also the node that represents the source document; this should not be confused with the root element, which is the element at the top of the hierarchy of elements. Figure 5-7 shows the node tree for a sample XML document. Note that although the XML declaration in the first line of the document is not treated as a node, the other lines of code in the document have corresponding entries in the node tree.

**Figure 5-7** A sample node tree



The relationship among the nodes in a node tree is based on a familial structure. A node that contains other nodes is called a **parent node**, and the nodes contained in a parent node are called **child nodes**. Nodes that share a common parent are called **sibling nodes**. Note that a node can have only one parent. As you progress further down the tree, any node found at a level below another node is referred to as a **descendant** of that node. The node at the top of the branch is referred to as the **ancestor** of all nodes that lie beneath it. In the sample document shown in Figure 5-7, the portfolio node is the parent of the child nodes author, date, time, and stock. Further down the node tree is the sName node, which is a descendant of both the portfolio and stock nodes. The root node has only two child nodes—the portfolio node and the processing instruction that links the XML document to the stock.xsl style sheet.

## Absolute and Relative Location Paths

One of the functions of XPath is to translate this hierarchical structure into an expression called a **location path** that references a specific node or node set from the source document. You can think of a location path as the directions that lead you through the node tree.

The location path can be written in either absolute or relative terms. An **absolute path** is a path that always starts from the root node and descends down through the node tree to a particular node or node set. The root node is identified by an initial forward slash (/), and then each level down the tree is marked by additional forward slashes. An absolute path that extends down the node tree from the root node has the following general form:

/child1/child2/child3/...

where *child1*, *child2*, *child3*, and so forth are the descendants of the root node. For example, the absolute path to the sName node from Figure 5-7 is

/portfolio/stock/sName

starting from the root node through the portfolio element and then the stock element within that and finally the sName element within the stock element. This particular

location path returns a node set that references all of the `sName` element nodes in the source document. Note that element nodes are identified by the element name.

A large document can have a long and complicated hierarchy. In order to avoid listing all of the levels of the node tree, you can use a double forward slash (`//`) with the syntax

```
//descendant
```

where `descendant` is the name of the descendant node. For example, the path

```
//sName
```

returns a node set containing all of the `sName` element nodes in the document, no matter where they're located.

Absolute paths can be long and cumbersome for large and complicated node trees. Therefore, most locations are written using **relative paths** in which the location path starts from a particular node (not necessarily the root node) called the **context node**. Rather than working through a path that navigates through the entire node tree, the XSLT processor needs only to work with a fragment of the tree. Figure 5-8 describes some of the common relative path expressions in XPath.

**Figure 5-8**

### XPath expressions for relative paths

Relative Path	Description
.	Refers to the context node itself
..	Refers to the parent of the context node
<code>child</code>	Refers to the child of the context node named <code>child</code>
<code>child1/child2</code>	Refers to the <code>child2</code> node, a child of the <code>child1</code> node beneath the context node
<code>../sibling</code>	Refers to a sibling of the context node named <code>sibling</code>
<code>./descendant</code>	Refers to a descendant of the context node named <code>descendant</code>
<code>.../..</code>	Refers to the parent of the parent of the context node

For example, if the `portfolio` element shown in Figure 5-7 is the context node, then the relative path that starts from the `portfolio` element and goes to the `sName` element has the path expression

```
stock/sName
```

Figure 5-9 provides other examples of the relative paths in the node tree from Figure 5-7 using the `stock` element as the context node. Take some time to study these expressions because relative paths are an important part of XPath.

**Figure 5-9**

### Resolving an XPath expression relative to a context node

Context Node	Relative Path	Description
stock	.	Refers to the <code>stock</code> element
	..	Refers to the <code>portfolio</code> element, the parent of the <code>stock</code> element
	<code>sName</code>	Refers to the <code>sName</code> element, a child of the <code>stock</code> element
	<code>../date</code>	Refers to the <code>date</code> element, a sibling of the <code>stock</code> element
	<code>./sName</code>	Refers to all descendent elements of the <code>stock</code> element named <code>sName</code>
	<code>.../..</code>	Refers to the parent of the parent of the <code>stock</code> element (in this case the root node)

### Identifying Nodes with Location Paths

#### For Absolute Paths

- To create an absolute reference to a node, use the location path expression  
`/child1/child2/child3/...`  
where `child1`, `child2`, `child3`, and so on are descendants of the root node.

#### For Relative Paths

- To reference a node without regard to its location in the node tree, use the expression  
`//descendant`  
where `descendant` is the name of the descendant node.
- To reference the context node, use  
`.`  
• To reference the parent of the context node, use  
`..`  
• To reference a child of the context node, use  
`child`  
where `child` is the name of the child node.
- To reference a sibling of the context node, use  
`../sibling`  
where `sibling` is the name of the sibling node.

XPath supports the wildcard character ( `*` ) to reference nodes of any type or name. For example, the XPath expression

`/portfolio/*`

is an absolute reference that references all of the child nodes of the `portfolio` element. To select all of the nodes in the node tree, you can use the path

`//*[1]`

In this expression, the `( *)` symbol matches any node, and the `( // )` symbol sets the scope of the path to include all of the descendants of the root node.

## Text, Comment, and Process Instruction Nodes

As you've seen, element nodes are referenced using the element's name. To reference a **text node**, which is the text contained within an element, you use the XPath expression:

`text()`

For example, the following location path returns a node set that contains all of the text nodes within the `sName` element node:

`//sName/text()`

Note that, because there are no nodes for character or entity references, if the element text contains an entity or character reference, then that reference is resolved by the XSLT processor before the text node is created.

Comments and processing instruction nodes can be referenced using the XPath expressions

```
comment()
```

and

```
processing-instruction()
```

Thus, the following location path returns a node set of all of the comments in the source document:

```
//comment()
```

There is usually little need for referencing comments or processing instructions in the source document until you get to more advanced XSLT applications.

You've barely scratched the surface of all that can be done with XPath. As you continue to work on XSLT style sheets in this and future tutorials, you'll return to XPath periodically to explore the various facets of this language. However, what you've learned is enough to start adding content to Rafael's style sheet that he will use to generate his stock market report.

## Introducing XSLT Templates

The basic building block of an XSLT style sheet is the template. A **template** is a collection of styles that are applied to a specific node set within the source document. The general syntax of an XSLT template is

```
<xsl:template match="node set">  
    styles  
</xsl:template>
```

where *node set* is an XPath expression that references a node set from the source document and *styles* are the XSLT styles applied to those nodes. An XSLT style sheet will usually have several templates with each template matching a particular node set. Templates simplify the process of creating a style sheet because the programmer only needs to write a style for a single set of nodes rather than the entire node tree. The final style sheet is therefore the sum total of individual templates.

### The Root Template

The fundamental template in the XSLT style sheet is the root template, which defines styles for the source document's root node. Because the root node refers to the source document itself (and *not* the root element), the root template sets the initial styles for the entire result document. The syntax for the root template is

```
<xsl:template match="/">  
    styles  
</xsl:template>
```

Note that the value of the *match* attribute is set to "/", which is the XPath expression for the root node. The root template can be located anywhere between the opening and closing *<xsl:stylesheet>* tags of the XSLT document. However, it is customary to put the root template at the top of the document, directly after the opening *<xsl:stylesheet>* tag.

## REFERENCE

**Creating a Template**

- To create an XSLT template that matches a specified node set from the source document, insert

```
<xsl:template match="node set">
    styles
</xsl:template>
```

where *node set* is an XPath expression that references the node set and *styles* are the XSLT styles defined for those node(s).

- To create a template for the root node, enter

```
<xsl:template match="/">
    styles
</xsl:template>
```

You add a root template to the stock.xsl file now.

**To create the root template:**

- 1. Return to the **stock.xsl** file in your text editor.
- 2. Within the **stylesheet** element, insert the following content:

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

Figure 5-10 shows the initial code for the root template.

**Figure 5-10**

**Inserting the template element**

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

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

</xsl:stylesheet>
```

Next you'll insert XSLT elements into the root template that will generate the contents of the result document.

## Literal Result Elements

Content is written to the result document through the use of XSLT elements and literal result elements. An **XSLT element** is any element that is part of the XSLT vocabulary. An XSLT element will usually contain instructions to the processor regarding how to interpret and render the contents from the source document. XSLT elements must be placed within the XSLT namespace, usually with the namespace prefix *xsl*.

A literal result element is any element that is not part of the XSLT vocabulary but is sent directly into the result document as raw text. For example, any HTML tags in a style sheet are considered a literal result because they are ignored by XSLT processors and written into the result document.

Rafael wants to create a web page based on the contents of the stock.xml file. He provides you with the following initial HTML code for his document:

```
<html>
  <head>
    <title>Portfolio Stocks</title>
    <link href="stock.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <header>
      <h1>Chesterton Financial</h1>
      <h2>Portfolio Stocks</h2>
    </header>
  </body>
</html>
```

### TIP

Even though HTML tags are treated as text, they still must follow XML syntax for well-formedness to be accepted by the XSLT processor.

All of the HTML elements in this code sample are literal result elements because they do not involve any of the elements associated with XSLT. The XSLT processor will write the HTML code directly into the result document without modification, creating an HTML document in the process.

#### To insert the HTML tags:

- 1. Within the root template, insert the following content immediately following the `<xsl:template>` tag:

```
<html>
  <head>
    <title>Portfolio Stocks</title>
    <link href="stock.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <header>
      <h1>Chesterton Financial</h1>
      <h2>Portfolio Stocks</h2>
    </header>
  </body>
</html>
```

Figure 5-11 highlights the initial code from the root template.

Figure 5-11

#### Literal result elements placed within the root template

HTML code to be written directly to the result document

```
<xsl:template match="/">
  <html>
    <head>
      <title>Portfolio Stocks</title>
      <link href="stock.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
      <header>
        <h1>Chesterton Financial</h1>
        <h2>Portfolio Stocks</h2>
      </header>
    </body>
  </html>
</xsl:template>
```

- 2. Save your changes to the file.

Note that Rafael has already created an external CSS style sheet named stock.css to format the appearance of his web page. If you want to review CSS usage, you can examine the contents of stock.css to see how the headings are formatted.

You've entered the initial code that you want written to the result document, but how does the processor know to create an HTML file based on this code rather than an XML file or a basic text file? You can control the type of file created by specifying the output method.

## Defining the Output Format

By default, an XSLT processor will create a result document as an XML file. There are two exceptions to this default behavior:

- If the root element in the result document is the `html` element, then the result document will be created as an HTML file.
- If the root element in the result document is the `html` element and it is placed in the XHTML namespace, `http://www.w3.org/1999/xhtml`, then the result document will be created as an XHTML file.

To explicitly define the format of the result document, you can add the following `output` element to the style sheet:

```
<xsl:output attributes />
```

where `attributes` is the list of attributes that control how the processor writes the result document. The `output` element should be placed directly after the opening `<xsl:stylesheet>` tag before any templates. Figure 5-12 describes the different attributes associated with this element.

**Figure 5-12**

**Attributes of the output element**

Attribute	Description
<code>method="xml html text"</code>	Defines the output format as xml (the default), html, or text
<code>version="number"</code>	Specifies the version of the output
<code>encoding="text"</code>	Specifies the character encoding
<code>omit-xml-declaration="yes no"</code>	Specifies whether to omit an XML declaration in the first line of the result document
<code>standalone="yes no"</code>	Specifies whether a standalone attribute should be included in the output and sets its value
<code>doctype-public="text"</code>	Sets the URI for the public identifier in the <code>&lt;!DOCTYPE&gt;</code> declaration
<code>doctype-system="text"</code>	Sets the system identifier in the <code>&lt;!DOCTYPE&gt;</code> declaration
<code>cdata-section-elements="list"</code>	Specifies a list of element names whose content should be output in CDATA sections
<code>indent="yes no"</code>	Specifies whether the output should be indented to better display its structure
<code>media-type="mime-type"</code>	Sets the MIME type of the output

For example, to create a file based on the strict XHTML DTD, you would include the following `output` element as part of your style sheet:

```
<xsl:output method="xml"
    indent="yes"
    encoding="UTF-8"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN" />
```

### TIP

Set the `indent` attribute to yes to make the code in your result document easier to read.

When the XSLT processor generates the result document it automatically adds the following XML declaration and a DOCTYPE to the beginning of the file, creating a XHTML file that can be validated against the strict XHTML DTD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

If you need to only to create an HTML file but do not need to have it validated, you can employ the following `output` element:

```
<xsl:output method="html"
    indent="yes"
    encoding="UTF-8"
    version="4.0" />
```

and the processor will generate code that is compliant with HTML 4.0, but the file will include neither an XML declaration nor a DOCTYPE.

For some applications, the result document will require only XML elements and attributes but not an XML declaration. Because it lacks an XML declaration, the result document is not a true XML document but rather an **XML fragment** containing part of a full XML document. XML fragments are created by including the following `omit-xml-declaration` attribute in the `output` statement:

```
<xsl:output method="xml"
    version="1.0"
    omit-xml-declaration="yes" />
```

Finally, to create documents that contain only text values, use the following `output` element with the `method` attribute set to "text":

```
<xsl:output method="text" />
```

This format could be used for creating Rich Text Format (RTF), which is a format supported by most word processors. To create an RTF file, you insert code for the RTF file into the style sheet. XSLT processors then pass the code through as text, without checking the document for well-formedness or validity or added XML tags.

Rafael's document will be written in the language of HTML5. He wants the file to conform to valid HTML5 standards, which include the presence of a DOCTYPE statement but do not include an XML processing instruction. To create an HTML5 file, you would use the following `output` element with the value of the `doctype-system` attribute set to "about:legacy-compat"

```
<xsl:output method="html"
    doctype-system="about:legacy-compat"
    encoding="UTF-8"
    indent="yes" />
```

You add this element now to Rafael's style sheet.

### To specify the output format of the result document:

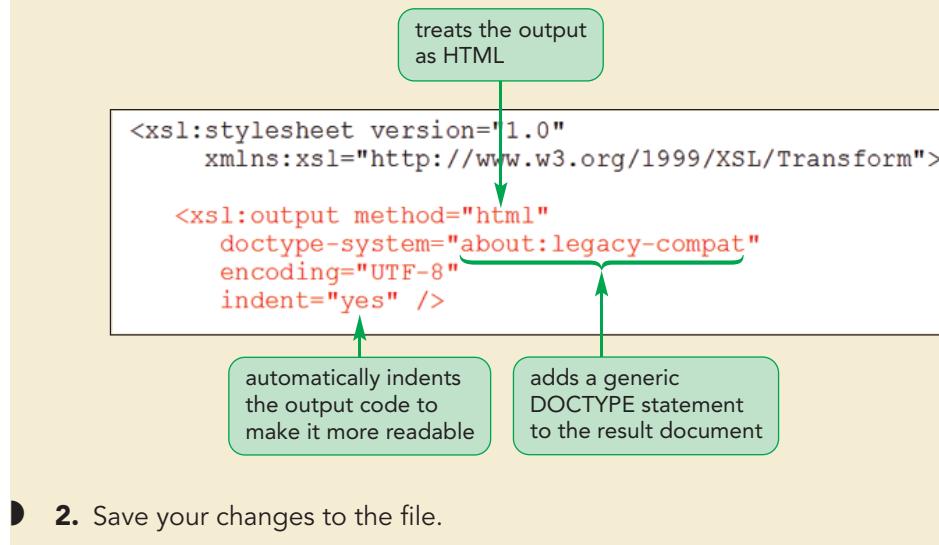
- 1. Insert the following code immediately after the opening `<xsl:stylesheet>` element located at the top of the stock.xsl file:

```
<xsl:output method="html"
    doctype-system="about:legacy-compat"
    encoding="UTF-8"
    indent="yes" />
```

Figure 5-13 highlights the `output` element to create a result document in HTML5 format.

Figure 5-13

### Setting the output format to HTML5



- 2. Save your changes to the file.

Now that you've specified an output method, you can view the initial result document generated by the XSLT style sheet.

## Transforming a Document

### TIP

If your web page is not rendered properly, you should always view the underlying HTML to look for missing element tags or elements that are not well-formed.

The simplest way to view a web page generated by an XSLT 1.0 style sheet is to open the source document in your web browser. Because most browsers have a built-in XSLT processor, the browser will automatically apply the XSLT style sheet to the source file and display the HTML file as rendered by the browser. If you want to view the underlying HTML code that was generated by the XSLT style sheet, you must use the developer tools available on your browser.

To review your progress in developing a style sheet for Rafael's stock report, you'll use your browser to open the stock.xml file and view its content as transformed by the stock.xsl style sheet. You can then use the developer's tools provided by your browser to view the actual HTML code generated by the style sheet.

### To view the result document:

- 1. Use your browser to open **stock.xml** from the xml05 ▶ tutorial folder. Your browser should display a main heading with the text "Chesterton Financial" and a subheading with the text "Portfolio Stocks".

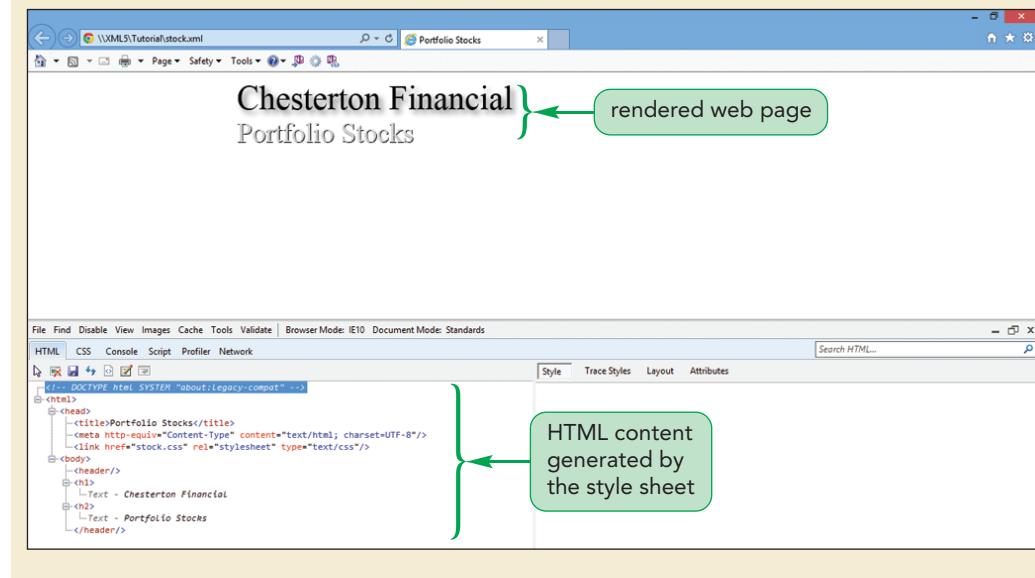
**Trouble?** If no page content appears you might have made a mistake in the XSLT file. Check your code against the code shown in Figure 5-11. If you are using Google Chrome, you will have to upload your files to a web server to view the result document.

- 2. Use the developer tools in your browser to view the HTML code generated by the style sheet. In Internet Explorer you open the developer tools by pressing the **F12** key. In Firefox and Google Chrome, you can view the HTML code by pressing **Ctrl+Shift+I**. For other web browsers, check the browser's online help for information on displaying the developer tools.

Figure 5-14 shows the appearance of the web page and the underling HTML code written by the style sheet as viewed under Internet Explorer.

Figure 5-14

### Viewing the result document within Internet Explorer



## Running Transformations Using Saxon

Another way to view the result document is to generate the document as a separate file using an XSLT processor. While it is not necessary to generate this file, the advantage of creating a separate file is that you can easily review the code generated by the style sheet and locate any errors in the output. You can also share the result document with other users who might not have access to an XSLT processor. A disadvantage is that every time you make a change to the style sheet you have to recreate the result file.

### TIP

You can also review the installation instructions in Appendix E.

In the steps that follow, you perform the transformation using the Saxon XSLT processor in Java command line mode. The free home edition of the Saxon XSLT processor can be downloaded from [saxon.sourceforge.net](http://saxon.sourceforge.net). To use Saxon you must also have Java or the .NET framework installed on your computer. You can download these programming environments freely from [www.java.com](http://www.java.com) or [www.microsoft.com/net/downloads](http://www.microsoft.com/net/downloads). The most current installation instructions for these programs are provided on the websites.

Once you have installed and configured both Java and Saxon on your computer, you can apply a transformation in Saxon Java command line mode by running the following command within a command prompt window:

```
java net.sf.saxon.Transform -s:source -xsl:style -o:output
```

where *source* is the XML source file, *style* is the XSLT style sheet file, and *output* is the result file. You can omit the *-s:* and *-xsl:* prefixes provided that the source and style

sheet file are listed prior to the output file and other Saxon command line parameters. For example, the following command applies the style sheet transformation from the stock.xsl file to the stock.xml file, storing the result document in the portfolio.html file:

```
java net.sf.saxon.Transform stock.xml stock.xsl -o:portfolio.html
```

If you are using Saxon on the .NET platform, the equivalent command line is

```
Transform -s:source -xsl:style -o:output
```

where, once again, you can omit the `-s:` and `-xsl:` prefixes provided the source and style sheet file are listed prior to the output file and other Saxon command line parameters. Thus, the same transformation under Saxon on the .NET platform is entered as

```
Transform stock.xml stock.xsl -o:portfolio.html
```

You use an XSLT processor now to create the result document in the portfolio.html file. If you have access to an editor other than Saxon using Java or the .NET platform, review the steps that follow and apply the corresponding commands available from your own editor to the task of generating a result document.

### To create the result document using Saxon:

- 1. If you are using Saxon in Java command line mode, go to the xml05 ▶ tutorial folder and run the following command from within a command window:

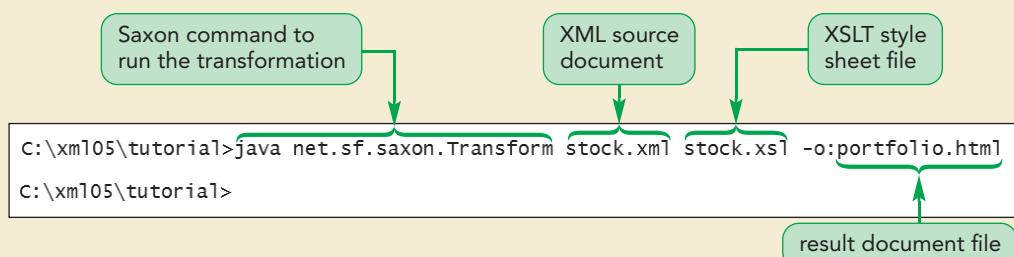
```
java net.sf.saxon.Transform stock.xml stock.xsl -o:portfolio.html
```

otherwise, use the commands appropriate to your XSLT processor to run the transformation and generate the result document.

Talk to your instructor or computer resource person for help in installing and running Saxon or any XSLT processor on your system. If you do not have access to an XSLT processor other than your web browser, you can review these steps for future reference. Note: You do not need a standalone processor to complete the remaining tasks in this tutorial.

Figure 5-15 shows the appearance of the Saxon command as it appears on a Windows command line.

**Figure 5-15** Running a transformation using Saxon in Java command line mode



**Trouble?** If Saxon returns the warning that you are running an XSLT 1 style sheet in an XSLT 2 processor, you can safely ignore the warning. Most of XSLT 2.0 is backward compatible with XSLT 1.0.

- 2. Open the **portfolio.html** file from the `xml05 ▶ tutorial` folder in a text editor and verify that the code matches that shown in Figure 5-16.

Figure 5-16

Result document code

```
<!DOCTYPE html
    SYSTEM "about:legacy-compat">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

        <title>Portfolio Stocks</title>
        <link href="stock.css" rel="stylesheet" type="text/css">
    </head>
    <body>
        <header>
            <h1>Chesterton Financial</h1>
            <h2>Portfolio Stocks</h2>
        </header>
    </body>
</html>
```

- 3. Close your text editor.

When you viewed the contents of the `portfolio.html` file, you may have noticed that the XSLT processor added the following extra line to the document:

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

The `meta` element makes it clear to any application opening this file what type of content the file contains and how the characters are encoded.



PROSKILLS

### Problem Solving: Debugging your Style Sheet

When you start creating your own style sheets you will invariably encounter a transformation that results in an error in which no code is sent to the result document or the code that is sent is missing key features and values. To debug your style sheets you should look for these common sources of error:

- **Attach the Style Sheet.** If your transformation is not being applied to the source document, confirm that you have attached the style sheet using the `<?xml-stylesheet>` processing instruction.
- **Validate.** XSLT files have to adhere to well-formed and valid XML. Test both the source document and the XSLT file for errors in structure or content before running your transformation.
- **Typos.** The names of elements and attributes in your XPath expressions must exactly match the names of elements and attributes in the source document, including upper- and lowercase letters.
- **Context matters.** Many errors occur because of mistakes in an XPath expression. Make sure your XPath expression makes sense based on the location of the context node.
- **Namespace Issues.** Make sure that all of your elements belong to the correct namespace and that the namespace has the correct prefix and URI.

If you still can't find the source of error, remember that most XML editors include debugging tools to locate errors in code or style sheet structure. Invest in an editor with a good debugger to quickly find the source of your mistakes.

At this point, you have not placed any content from the source document into the result document. All of the content of the result document has been literal result elements. In the next session, you'll learn how to retrieve data values from the source file and display them in the result document.

**REVIEW****Session 5.1 Quick Check**

1. What are XSLT, XPath, and XSL-FO?
2. Provide the processing instruction to link an XML document to an XSLT style sheet named styles.xsl.
3. Using the node tree diagram in Figure 5-7, provide the absolute path to the author element node.
4. Using the node tree diagram in Figure 5-7, if the context node is the author element node, provide the relative path to the sName element node.
5. Provide an XPath expression to return a node set of all author elements, regardless of their location in the source document.
6. What is a literal result element?
7. Provide the XSLT element to specify that the result document should be a transitional XHTML file with a system DTD sent to “<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>” and the public DTD set to “`-//W3C//DTD XHTML 1.0 Transitional//EN`”. Include an attribute to have the XSLT processor indent the output to make it easier to read.

## Session 5.2 Visual Overview:

```

<body>
  <header>
    <section>
      Last Updated:
      <xsl:value-of select="portfolio/date" />
      at
      <xsl:value-of select="portfolio/time" />
    </section>

    <h1>Chesterton Financial</h1>
    <h2>Portfolio Stocks</h2>
  </header>

  <section>
    <xsl:apply-templates select="portfolio/stock" />
  </section>
</body>

```

The **xsl:value-of** element writes the value of the selected nodes.

The **xsl:apply-templates** element applies a template for the selected nodes.

# Chesterton Financial

## Portfolio Stocks

This is the value of the sName element and the @symbol attribute.

This is the value of the description element.

### Boeing Company (BA)

The Boeing Company engages in the design, development, manufacture, sale, and support of commercial jetliners, military aircraft, satellites, missile defense, human space flight, and launch systems and services worldwide. It also offers aviation services support, aircraft modifications, spares, training, maintenance documents, and technical advice to commercial and government customers. Its financing portfolio consists of equipment under operating leases, finance leases, notes and other receivables, assets held for sale or re-lease, and investments. The Boeing Company was founded in 1916 and is based in Chicago, Illinois.

# Applying Templates to Document Nodes

The **match** attribute of the `xsl:template` element is used to design the output format for the specified node set.

Attribute nodes are referenced using the path expression `@att` where `att` is the name of the attribute.

```

<xsl:template match="stock">
    <article>

        <h1>
            <xsl:value-of select="sName" />
            (<xsl:value-of select="sName/@symbol" />)
        </h1>

        <xsl:apply-templates select="today" />

        <p>
            <xsl:value-of select="description" />
        </p>
        <hr />

    </article>
</xsl:template>

```

Element nodes are referenced using the element name.

Last Updated: 4/17/2017 at 13:17

This is the value of the date and time elements.

Current Stock Values				
Current	Open	High	Low	Volume
125.68	125.68	125.72	124.81	6274500

## Extracting Element Values

In the last session you worked with Rafael to create the initial XSLT style sheet for the stock report. Because the style sheet wrote specific lines of HTML code, it was not any more efficient than writing the HTML code yourself without a style sheet. However, Rafael wants the style sheet to generate HTML code that is based on the data values found within the source document and that will change if those data values change. Thus you need to retrieve data values from the source document in the result document.

To display a data value from a node in the source document, XSLT employs the following `value-of` element:

```
<xsl:value-of select="node" />
```

where `node` is a location path that references a node from the source document's node tree. For element nodes that contain only text, the node value is simply that text string. If the element node contains child elements in addition to text content, the text in those child nodes appears as well.

### REFERENCE

#### Inserting a Node's Value

- To display a node value, use:

```
<xsl:value-of select="node" />
```

where `node` is an XPath expression that identifies the node from the source document's node tree.

To see how this works, return to Rafael's proposed sketch of his web page, shown earlier in Figure 5-5. At the top of the page, Rafael wants to display the date and time values from the source document as follows:

`Last Updated: date at time`

where `date` and `time` are values taken from the date and time elements. To display these values you add the following `value-of` elements to the root template of the style sheet using location paths that point to the date and time elements in the source document:

```
Last Updated:  
<xsl:value-of select="portfolio/date" />  
at  
<xsl:value-of select="portfolio/time" />
```

Note that these location paths are relative and not absolute paths. They are expressed relative to the context node, which, in this case, is the root node because these styles will be placed within the root template. By default, any relative path is always relative to the node that the template matches.

#### To display the values of the date and time elements:

- 1. If you took a break after the previous session, make sure the `stock.xsl` file is open in your text editor.

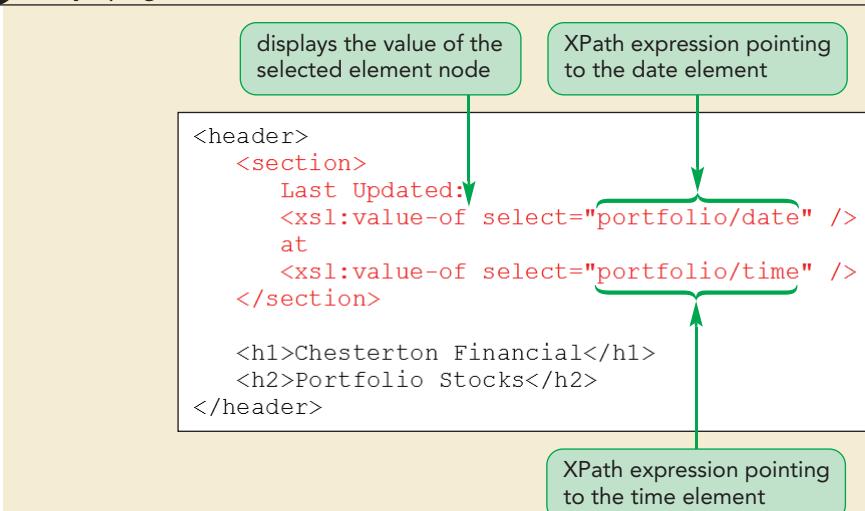
2. Insert the following code within the header element inside of the root template:

```
<section>
    Last Updated:
    <xsl:value-of select="portfolio/date" />
    at
    <xsl:value-of select="portfolio/time" />
</section>
```

Figure 5-17 shows the location of the style code within the root template.

Figure 5-17

### Displaying the value of the date and time elements



3. Save your changes to the file.  
4. Using either your browser (if it contains a built-in XSLT processor) or an XSLT processor, generate the revised result document.

Note: If you use the editor, follow the techniques described in the last session to update the contents of the portfolio.html file. You will need to update the portfolio.html file each time if you want to keep the file current.

Figure 5-18 shows the appearance of the revised web page with the date and time values retrieved from the stock.xml file.

Figure 5-18

### Revised result document

Chesterton Financial  
Portfolio Stocks

date and time value taken from the stock.xml file

Last Updated: 4/17/2017 at 13:17

The stock.xml file contains the names of 15 stocks. Next, you'll display the name of each stock as an h1 heading in the document and placed within its own article by writing the following HTML5 code to the result document:

```
<section>
  <article>
    <h1>stock name</h1>
  </article>
</section>
```

where *stock name* is the name of the stock, as retrieved from the sName element. To generate this code you'll add the following code to the root template:

```
<section>
  <article>
    <h1>
      <xsl:value-of select="portfolio/stock/sName" />
    </h1>
  </article>
</section>
```

If you're unclear about the location path used in this expression, refer to the source document or to the tree diagram shown earlier in Figure 5-3. You'll add this code now to the stock.xsl style sheet.

### To display a stock name:

- 1. Return to the **stock.xsl** file in your text editor.
- 2. Insert the following code above the closing `</body>` tag within the root template:

```
<section>
  <article>
    <h1>
      <xsl:value-of select="portfolio/stock/sName" />
    </h1>
  </article>
</section>
```

Figure 5-19 shows the code used to display the names of the stock in the portfolio.

**Figure 5-19**

**Code to display the stock name**

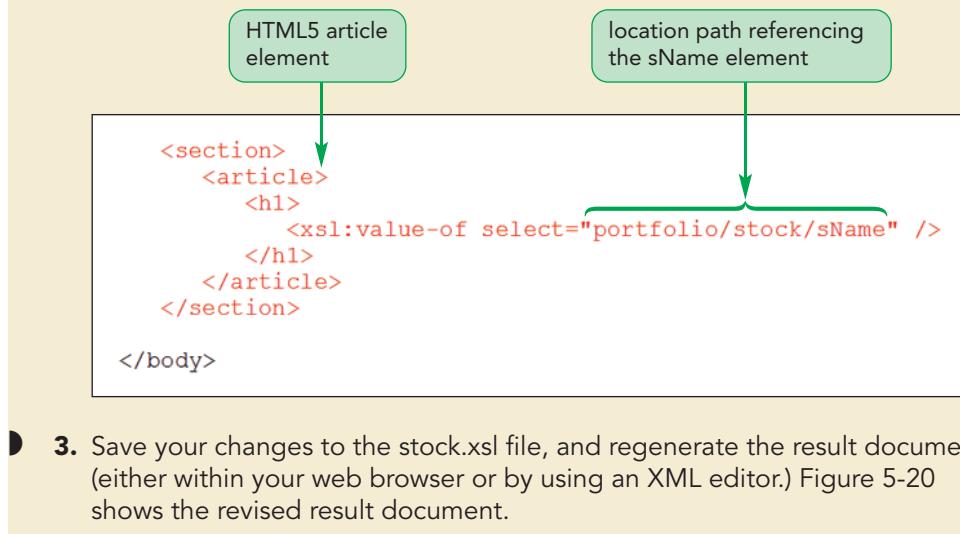


Figure 5-20

## Name of the first stock in the portfolio

**Chesterton Financial**

## Portfolio Stocks

Boeing Company

Last Updated: 4/17/2017 at 13:17

only the first stock is listed

The first stock name listed in the stock.xml file appears in the document, but where are the other 14 stocks? In XSLT 1.0, when the location path returns more than one possible node, the `value-of` element will display only the value of the first node in that node set (in XSLT 2.0 the values of all of the matching nodes are displayed.)

Thus, to list the names of the stocks in the portfolio, you'll revise the style sheet to go through each stock element in the node tree.

**INSIGHT****Using XSLT with HTML Markup Tags**

XSLT can be used to transform data from documents written in any XML vocabulary, including RSS news feeds. Some RSS news feeds will store news items within HTML tags placed inside CDATA sections. Thus, one challenge for transforming data from a news feed is to retain the HTML tag structure in the result document. For example, the following code from an RSS feed uses a CDATA section to store the heading for a news story within an `h1` element used for marking main web page headings.

```
<description><![CDATA[<h1>News Story</h1>
]]></description>
```

When the XSLT processor encounters this HTML code, it will replace all of the HTML tags with escape characters, writing the following text to the result document:

```
&lt;h1&gt;News Story&lt;/h1&gt;
```

and losing the `h1` heading tag in the process.

To overcome this problem, you can disable the replacement of HTML tags with escape characters using the `disable-output-escaping` attribute. The following code shows how to write the HTML code in the `description` element directly without escaping:

```
<xsl:value-of select="description" disable-output-
escaping="yes" />
```

Now the processor will send the HTML code directly to the result document as it is stored in the CDATA section, including all of the HTML markup tags. This technique is particularly useful when you need to place large sections of HTML code in the result document without losing the HTML tags in the process.

## Using the `for-each` Element

If there are multiple nodes that match the location path, you can create a style for each matching node using the following `for-each` instruction:

```
<xsl:for-each select="node set">
    styles
</xsl:for-each>
```

where `node set` is a location path that returns a set of one or more nodes and `styles` are the XSLT styles applied to each node in the node set. Thus, to apply the same style to each `sName` element in the `stock.xml` file, you apply the following code:

```
<section>
    <xsl:for-each select="portfolio/stock">
        <article>
            <h1>
                <xsl:value-of select="sName" />
            </h1>
        </article>
    </xsl:for-each>
</section>
```

As the XSLT processor goes through the source document's node tree, it stops at each occurrence of a `portfolio/stock` node and writes the HTML code enclosing the value of the `sName` element. Note that the statement `<xsl:value-of select="sName" />` uses a relative path reference to point to the `sName` element because, in this situation, the context node is the `portfolio/stock` element node.

**REFERENCE**

### Applying a Style for each Node in a Node Set

- To apply a style to each occurrence of a node, use

```
<xsl:for-each select="node set">
    styles
</xsl:for-each>
```

where `node set` is an XPath expression that matches several nodes from the source document's node tree and `styles` are styles that are applied to each node matching that XPath expression.

Next, you use the `for-each` instruction to display all of the stock names from the source document.

### To display multiple stock names:

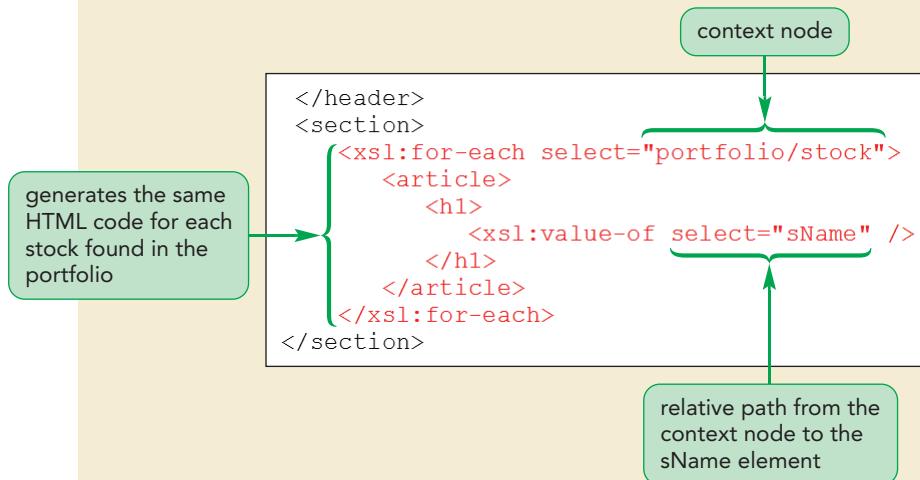
- 1. Return to the `stock.xsl` file in your text editor.
- 2. Replace the five lines of code within the `section` element containing the `<xsl:for-each select="portfolio/stock">` tag with the following:

```
<xsl:for-each select="portfolio/stock">
    <article>
        <h1>
            <xsl:value-of select="sName" />
        </h1>
    </article>
</xsl:for-each>
```

Figure 5-21 shows the application of the `for-each` instruction to display the value of each `sName` element.

Figure 5-21

Name of the first stock in the portfolio



- 3. Save your changes to the file and then regenerate the result document using either your web browser or an XML editor. Verify that all 15 stock names are now displayed in the document. The stock names are listed in the order they appear in the source document.

The `for-each` instruction is one way to apply styles to each node from a node set. A more versatile approach is to create a template.

## Working with Templates

You've already created a template for the source document's root node. However templates can be defined for any node set specified by an XPath expression. For example, a template that displays the value of the `sName` element could be entered as

```

<xsl:template match="stock">
  <article>
    <h1>
      <xsl:value-of select="sName" />
    </h1>
  </article>
</xsl:template>

```

### TIP

To display the value of the context node, use `<xsl:value-of select="." />` within the template.

This template matches every occurrence of the `stock` element in the source document, creating a style that displays the value of the `sName` element as an `h1` heading within an `article` element. The node specified in the `match` attribute sets the context for any location paths defined within the template. In this example, all location paths within the `stock` template are interpreted relative to the position of the `stock` element. Thus, the XPath expression `select="sName"` is interpreted by the XSLT processor as referencing "the `sName` element that is a child of the `stock` element."

Simply creating a template does not force the XSLT processor to use it with the result document. You must also indicate where you want that template applied.

## Applying a Template

To apply a template, use the following `apply-templates` instruction:

```
<xsl:apply-templates select="node set" />
```

where `node set` is a location path that references a node set in the source document. The XSLT processor then searches the XSLT style sheet for a template matching that node set. For example, the following code applies the template written for the `stock` element:

```
<xsl:template match="portfolio">
    <xsl:apply-templates select="stock" />
</xsl:template>
```

Because the context node for this template is the `portfolio` element, the `apply-templates` instruction matches the `stock` element nested within the `portfolio` element and the XSLT processor will apply any styles written for that particular node. However, the following template would have the same effect:

```
<xsl:template match="/">
    <xsl:apply-templates select="portfolio/stock" />
</xsl:template>
```

Here, the root from the source document is the context node and the `stock` template is applied for every occurrence of the `portfolio/stock` node set within the root node. In both cases, the XSLT processor searches the XSLT style sheet for a template that matches the `stock` node, no matter how the location path is specified by the `select` attribute in the `apply-templates` instruction. The template is then applied each time a matching node is found in the source document. This is why templates can be used in place of the `for-each` instruction.

### REFERENCE

#### Applying a Template

- To apply a template, use the XSLT instruction

```
<xsl:apply-templates select="node set" />
```

where `node set` is an XPath expression matching a node set from the source document.

You'll replace the `for-each` instruction with a template for the `stock` element and then apply that template in the style sheet.

#### To create and apply a template:

- 1. Return to the `stock.xsl` file in your text editor.
- 2. Delete the `for-each` construction from within the `section` element (all of the lines shown in red in Figure 5-21), and replace it with the following line:

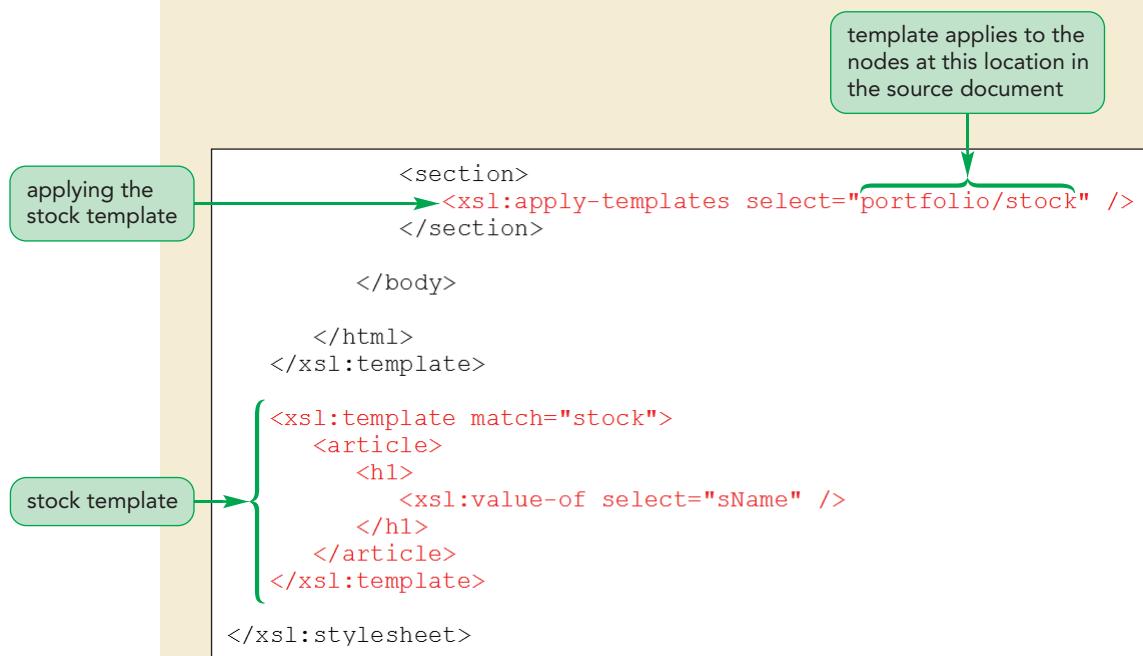
```
<xsl:apply-templates select="portfolio/stock" />
```

3. Insert the following code immediately above the closing `</xsl:stylesheet>` tag:

```
<xsl:template match="stock">
  <article>
    <h1>
      <xsl:value-of select="sName" />
    </h1>
  </article>
</xsl:template>
```

Figure 5-22 shows the revised file employing the use of templates in place of for-each instructions.

**Figure 5-22 Applying the stock template**



4. Save your changes to the file, and regenerate the result document, verifying that the result document once again shows the names of all 15 stocks in the portfolio.

Rafael wants you to add a paragraph to each article describing the stock and its history. This information is stored within the `description` element. So, next, you add the value of this element to the stock template displayed within a paragraph. You also add a horizontal rule after the paragraph using the `<hr />` tag.

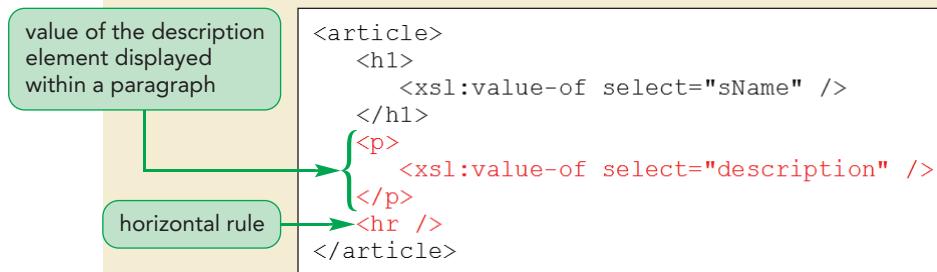
### To display a paragraph of descriptive text:

- 1. Return to the `stock.xsl` file in your text editor.
- 2. Directly above the closing `</article>` tag, insert the following:

```
<p>
  <xsl:value-of select="description" />
</p>
<hr />
```

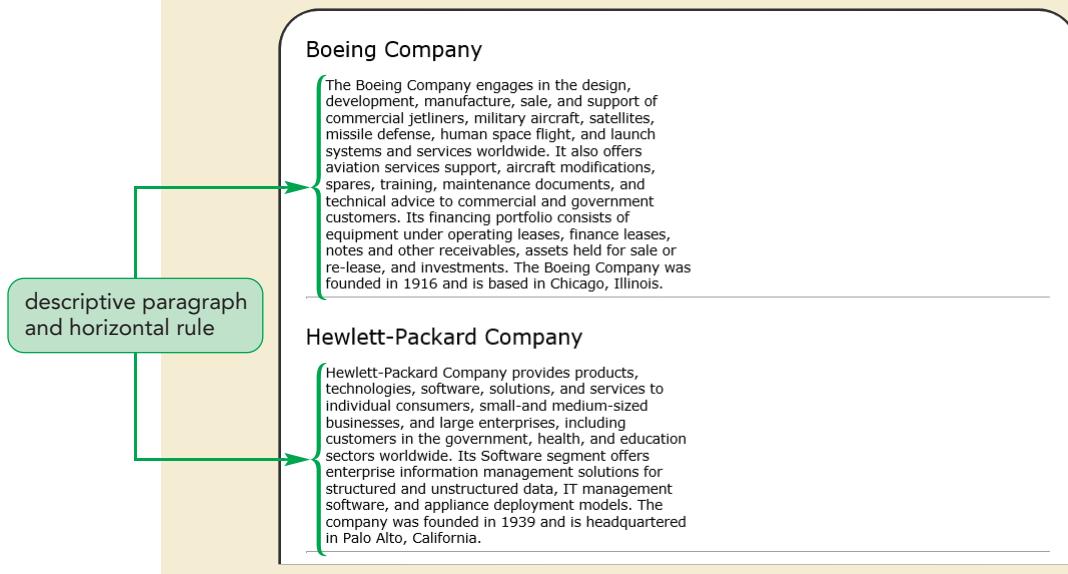
Figure 5-23 shows the placement of the paragraph tag and horizontal rule.

**Figure 5-23** Revised stock template



- 3. Save your changes to the file and regenerate the result document. As shown in Figure 5-24, a paragraph of descriptive text and a horizontal rule is included with the output for each company.

**Figure 5-24** List of stocks with descriptive paragraphs



Stocks are identified both by their stock name and their ticker symbol. For example, the stock symbol for the Boeing Company is “BA”. Rafael wants to include the ticker symbol in his report. To add that information, you’ll have to work with attribute nodes.

**INSIGHT****Applying Templates without a Select Attribute**

The `select` attribute is not required with the `<xsl:apply-templates>` tag. When the `select` attribute is omitted the `apply-templates` instruction automatically processes all of the children of the context node, including any text nodes. For example, if the stock element document contains the child elements

```
<stock>
  <sName>Boeing Company</sName>
  <category>Transportation</category>
</stock>
```

you can apply templates to each child element with

```
<xsl:template match="stock">
  <xsl:apply-templates select="sName" />
  <xsl:apply-templates select="category" />
</xsl:template>
```

Or, more efficiently you can omit the `select` attribute and use

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

Both the `sName` and `category` templates will be automatically applied by the processor, but the code will be more efficient and run more quickly by omitting the `select` attribute and using the `<xsl:apply-templates />` tag.

**Displaying Attribute Values**

The location paths you've seen so far have been concerned only with element nodes, but attributes can also be included in a location path using the XPath expression:

*node@attribute*

where `node` is an element node and `attribute` is the name of an attribute for that node. For example, the `sName` element has a single attribute named `symbol`. The absolute reference to this attribute is

`/portfolio/stock/sName/@symbol`

Attribute nodes can also be used in relative paths. Thus, if the `stock` element is the context node, then the relative path to the `symbol` attribute now becomes

`sName/@symbol`

Rafael wants the name of each stock to include the ticker symbol as follows:

`stock name (ticker symbol)`

where `stock name` is drawn from the `sName` element and `ticker symbol` is drawn from the `symbol` attribute of the `sName` element. The code to display these values is therefore

```
<xsl:value-of select="sName" />
(<xsl:value-of select="sName/@symbol" />)
```

You modify the `sName` template now, adding the value of the `symbol` attribute.

### To display the symbol attribute value:

- 1. Return to the **stock.xsl** file in your text editor.
- 2. Insert the following code immediately above the closing `</h1>` tag in the stock template:

```
(<xsl:value-of select="sName/@symbol" />)
```

Figure 5-25 highlights the code to display the value of the symbol attribute.

**Figure 5-25**

### List of stocks with descriptive paragraphs

- 3. Save your changes to the style sheet, and regenerate the result document. Verify that the ticker symbol for each stock appears in parentheses after the stock name.

Next, Rafael wants to display the current values of each stock. This information is stored as attributes of the `today` element. For example, the opening, high, low, current, and volume values of the Boeing Company stock are entered in the `stock.xml` file as:

```
<today open="125.68" high="125.72" low="124.81" current="125.68"
vol="6274500" />
```

Rafael wants this data displayed in a table. The HTML code for creating the table header is

```
<table>
  <thead>
    <tr>
      <th colspan="5">Current Stock Values</th>
    </tr>
    <tr>
      <th>Current</th>
      <th>Open</th>
      <th>High</th>
      <th>Low</th>
      <th>Volume</th>
    </tr>
  </thead>
</table>
```

Next, you insert this code into a new template matching the `today` element.

### To start creating the today template:

- 1. Return to the **stock.xsl** file in your text editor.
- 2. Go to the end of the file and directly before the closing `</xsl:stylesheet>` tag, insert the following template matching the `today` element:

```
<xsl:template match="today">
  <table>
    <thead>
      <tr>
        <th colspan="5">Current Stock Values</th>
      </tr>
      <tr>
        <th>Current</th>
        <th>Open</th>
        <th>High</th>
        <th>Low</th>
        <th>Volume</th>
      </tr>
    </thead>
  </table>
</xsl:template>
```

Figure 5-26 shows the complete code for the header row of the today table.

Figure 5-26

### Initial today template

```
<xsl:template match="today">
  <table>
    <thead>
      <tr>
        <th colspan="5">Current Stock Values</th>
      </tr>
      <tr>
        <th>Current</th>
        <th>Open</th>
        <th>High</th>
        <th>Low</th>
        <th>Volume</th>
      </tr>
    </thead>
  </table>
</xsl:template>

</xsl:stylesheet>
```

The table body will consist of a single row with each cell in the row displaying a different stock attribute. The HTML code for the table body is

```
<tbody>
  <tr>
    <td>current</td>
    <td>open</td>
    <td>high</td>
    <td>low</td>
    <td>volume</td>
  </tr>
</tbody>
```

where *current*, *open*, *high*, *low*, and *volume* are the corresponding attributes from the *today* element. Notice that the five table cells have essentially the same code aside from the value being displayed. It would be more efficient to write this code by applying the same template to the different attribute values. You can do this by combining different node sets into a single location path.

## Combining Node Sets

Multiple nodes sets can be combined into a single location path using the union ( | ) operator. For example, the expression

```
/portfolio/date | /portfolio/time
```

defines a location path that matches both the date and time elements nested within the portfolio element. Similarly, the expression

```
@open|@high|@low|@current|@vol
```

matches the open, high, low, current, and vol attributes. Thus, the following template matches all five attributes of the today element:

```
<xsl:template match="@open|@high|@low|@current|@vol">
    <td><xsl:value-of select="." /></td>
</xsl:template>
```

Note that the template displays the attribute value by using the XSLT element tag `<xsl:value-of select="." />` to return the value of the context node, which, in this case, is the value of the open, high, low, current, or vol attribute.

To display the attribute values as table cells, you would then apply the template as follows:

```
<tbody>
<tr>
    <xsl:apply-templates select="@current" />
    <xsl:apply-templates select="@open" />
    <xsl:apply-templates select="@high" />
    <xsl:apply-templates select="@low" />
    <xsl:apply-templates select="@vol" />
</tr>
</tbody>
```

Next, you add this code to the today template to write the body of the today table and then apply it.

### To complete the today template:

- 1. In the stock.xls file, directly above the closing `</table>` tag in the today template, insert the following code:

```
<tbody>
<tr>
    <xsl:apply-templates select="@current" />
    <xsl:apply-templates select="@open" />
    <xsl:apply-templates select="@high" />
    <xsl:apply-templates select="@low" />
    <xsl:apply-templates select="@vol" />
</tr>
</tbody>
```

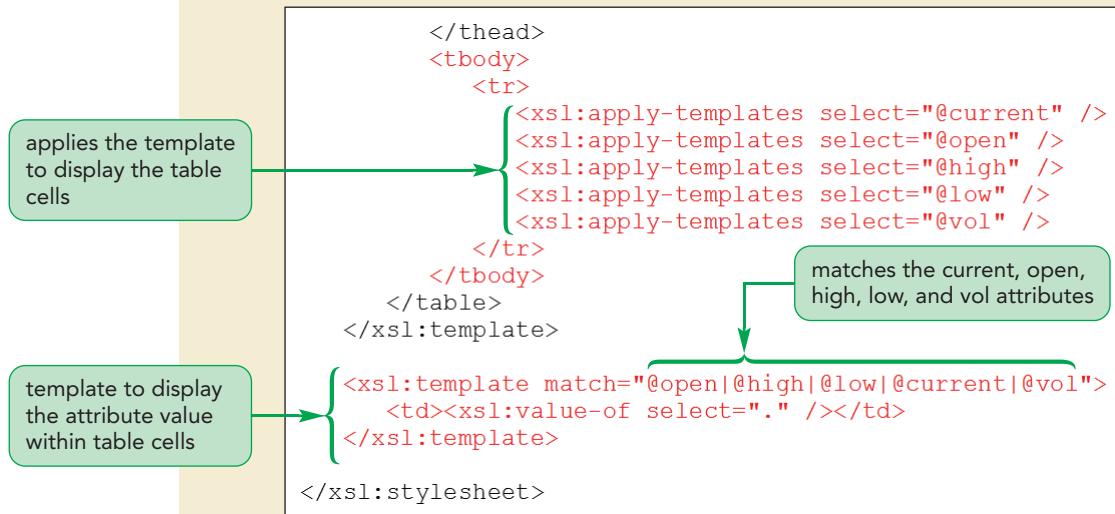
2. Directly above the closing `</xsl:stylesheet>` tag, insert the following code to create the template for the attribute values:

```
<xsl:template match="@open|@high|@low|@current|@vol">
  <td><xsl:value-of select=". " /></td>
</xsl:template>
```

Figure 5-27 shows the revised code.

**Figure 5-27**

### Template for displaying attribute values



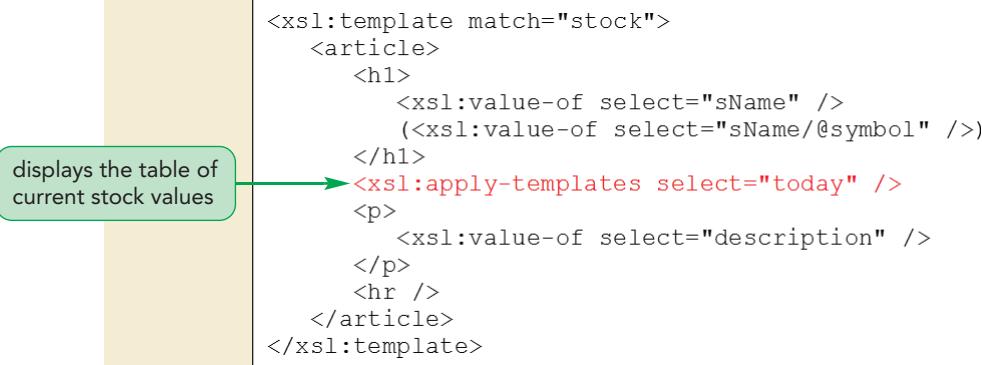
3. Next you'll apply the today template to display the table of current stock values for each stock listed in the result document.
4. Scroll up to the stock template and, directly after the closing `</h1>` tag and before the `<p>` tag, insert

```
<xsl:apply-templates select="today" />
```

Figure 5-28 highlights the `apply-templates` instruction in the today template.

**Figure 5-28**

### Applying the today template



5. Save your changes to the file and then regenerate the result document. The table of current stock values appears alongside the description of the stock, as shown in Figure 5-29.

Figure 5-29

Table of current stock values

### Boeing Company (BA)

The Boeing Company engages in the design, development, manufacture, sale, and support of commercial jetliners, military aircraft, satellites, missile defense, human space flight, and launch systems and services worldwide. It also offers aviation services support, aircraft modifications, spares, training, maintenance documents, and technical advice to commercial and government customers. Its financing portfolio consists of equipment under operating leases, finance leases, notes and other receivables, assets held for sale or re-lease, and investments. The Boeing Company was founded in 1916 and is based in Chicago, Illinois.

Current Stock Values				
Current	Open	High	Low	Volume
125.68	125.68	125.72	124.81	6274500

values of the current, open, high, low, and vol attributes

### INSIGHT

#### Built-In Templates

XSLT supports several **built-in templates** that specify how the values of different nodes are displayed, by default. For example, the following built-in template defines how the values of all text nodes and all attribute nodes from the source document are displayed:

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

For this built-in template to be invoked, the element and attribute nodes from the source document have to be assigned a template written by the programmer. Once such a template is assigned, the XSLT processor automatically applies this built-in template, rendering the text of the element or attribute within the result document.

Comments and processing instruction nodes have the following built-in template:

```
<xsl:template match="comment()|processing-instruction()" />
```

Because the purpose of this template is to keep comments and processing instructions from appearing in the result document, no values are ever sent to the result document. Note that this **template** element appears in a one-sided tag because it contains no content.

The most fundamental built-in template is the following:

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

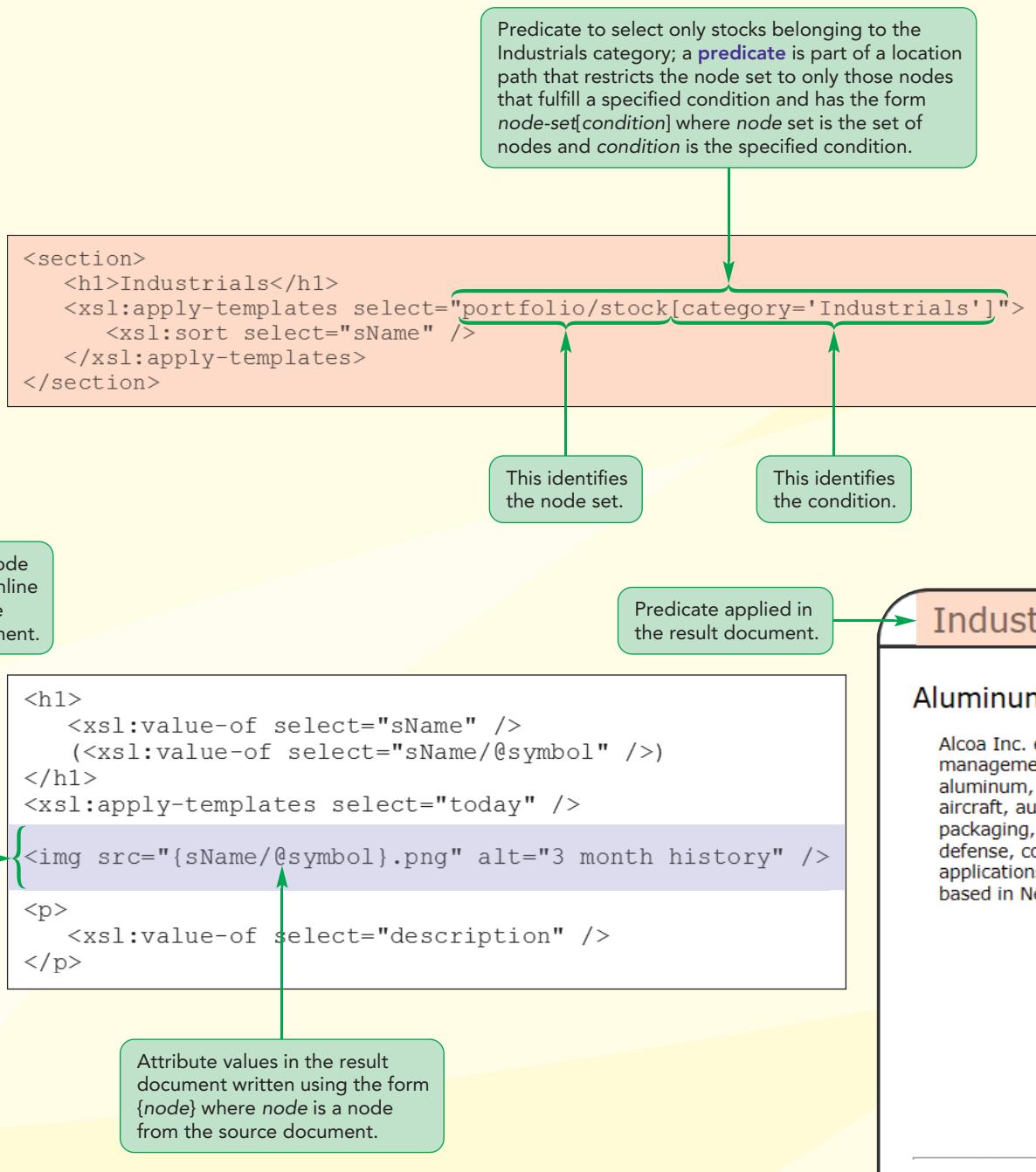
which applies to all element nodes and the root node. No **select** attribute is given for the **apply-templates** instruction, which causes the XSLT processor to locate all of the element nodes from the source document. The result is that the processor navigates the entire node tree, searching for templates to apply, including those written by the style sheet's author.

At this point, you've added all of the stock data that Rafael wants to show in the result document. In the next session, you'll insert attribute values and learn how to modify the appearance of the document by sorting the nodes of the node tree and creating conditional nodes.

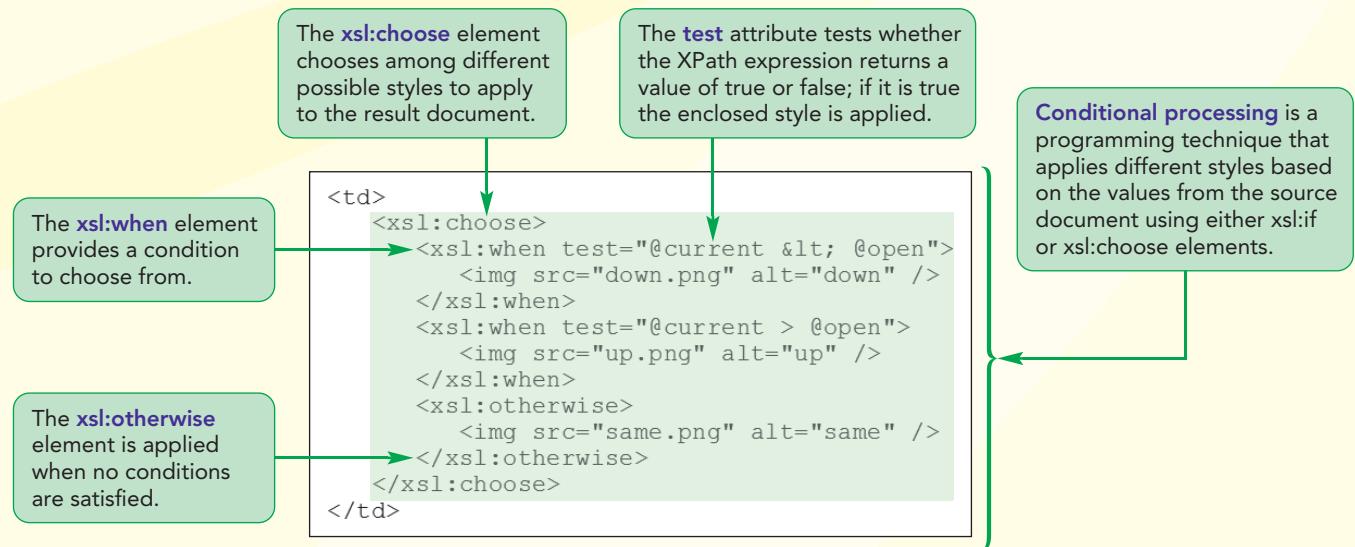
**REVIEW****Session 5.2 Quick Check**

1. Provide the command to display the value of the book element nested within the catalog element.
2. Provide the command to display the value of the context node.
3. Provide the command to display the value of the parent of the context node.
4. Provide the code to apply a template for the location path "catalog/books".
5. Provide the code to apply a template for either the "catalog/books" or "catalog/authors" node sets.
6. Provide the code to display the value of the ISBN attribute of the book element.
7. What is a built-in template?

## Session 5.3 Visual Overview:



# Applying a Conditional Expression



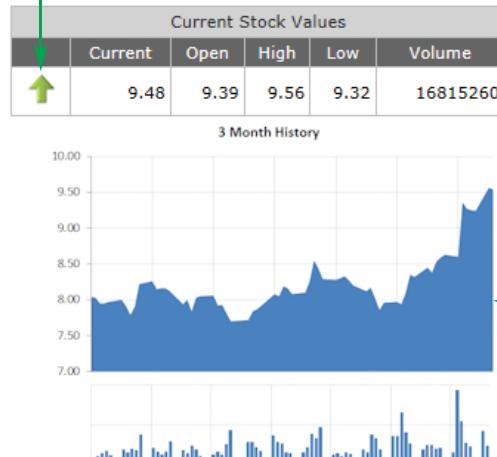
Is

Company of America (AA)

ges in the production and primary aluminum, fabricated alumina. Its products are used in mobiles, commercial transportation, mining and construction, oil and gas, consumer electronics, and industrial

coa Inc. was founded in 1888 and is located in New York.

The icon to be displayed is chosen using the `xsl:choose` element.



Inline image with its source determined by the value of the `symbol` attribute.

## Inserting a Value into an Attribute

### TIP

Curly braces cannot be nested within the attribute value.

In the last session, you learned how to write values into the elements of the result document. You can also use XSLT to write values in the attributes of those elements by enclosing an XPath expression within a set of curly braces using the general form:

```
<element attribute="{expression}">
```

where *element* is the name of the element written to the result document, *attribute* is the element's attribute, and *expression* is an XPath expression that sets the attribute's value. For example, the following code writes a hypertext link using the value of the link node as the URL of the link:

```
<a href="{link}">
    Stock Report
</a>
```

If the value of the link node contains the text "http://www.example.com/stockreports", the XSLT processor will generate the following HTML code:

```
<a href="http://www.example.com/stockreports">
    Stock Report
</a>
```

Rafael wants his web page to include graphic images charting stock values from the past three months. He has provided you with 15 image files—one for each stock in the portfolio. The image files are named *ticker.png* where *ticker* is the stock's ticker symbol, such as BA.png for the Boeing Company chart. Rafael wants the style sheet to write the following inline image tag for each stock in the portfolio:

```

```

Because the ticker symbol is an attribute of the *sName* element, you can write this image tag by enclosing the *sName/@symbol* path within curly braces for the *src* attribute in the following style sheet statement:

```

```

Thus, for the Boeing Company stock, which has the ticker symbol BA, the style sheet will write the following HTML inline image tag with the *src* attribute value equal to BA.png:

```

```

You revise the style sheet now to write this code for each stock in the portfolio.

### To write an attribute value:

- 1. Return to the **stock.xsl** file in your text editor and scroll down to the stock template.
- 2. Directly above the opening **<p>** tag, insert the following:

```

```

Figure 5-30 highlights the newly added code in the style sheet.

When writing a value to an XML attribute, you must enclose the XPath expression within curly braces.

Figure 5-30

## Writing an attribute value

displays an inline image of the stock's recent performance

```
<xsl:template match="stock">
  <article>
    <h1>
      <xsl:value-of select="sName" />
      (<xsl:value-of select="sName/@symbol" />)
    </h1>
    <xsl:apply-templates select="today" />
    
    <p>
      <xsl:value-of select="description" />
    </p>
    <hr />
  </article>
</xsl:template>
```

inserts the value of the symbol attribute

- 3. Save your changes to the file and then regenerate the result document. Figure 5-31 shows the appearance of the stock alongside the 3-month history chart.

Figure 5-31

## Stock information with the 3 year history chart

## Boeing Company (BA)

The Boeing Company engages in the design, development, manufacture, sale, and support of commercial jetliners, military aircraft, satellites, missile defense, human space flight, and launch systems and services worldwide. It also offers aviation services support, aircraft modifications, spares, training, maintenance documents, and technical advice to commercial and government customers. Its financing portfolio consists of equipment under operating leases, finance leases, notes and other receivables, assets held for sale or re-lease, and investments. The Boeing Company was founded in 1916 and is based in Chicago, Illinois.

Current Stock Values				
Current	Open	High	Low	Volume
125.68	125.68	125.72	124.81	6274500



inline image from the BA.png file

## Sorting Node Sets

Nodes are displayed in the result document in the same order in which they appear in the source document's node tree. To sort the nodes in a different order, you can apply the following `sort` instruction in the style sheet:

```
<xsl:sort select="node set" data-type="text|number|qname"
           order="ascending|descending"
           case-order="upper-first|lower-first"
           lang="language" />
```

where `node set` is an XPath expression that returns a set of nodes, the `data-type` attribute specifies the type of data to be sorted (text, number, or qname for qualified XML names), the `order` attribute defines whether to sort in ascending or descending order, the `case-order` attribute specifies whether uppercase or lowercase characters are

to be sorted first, and the `lang` attribute defines the language used to determine sort order. For example, the following `sort` instruction sorts the `sName` element in descending order with lowercase characters sorted prior to uppercase letters:

```
<xsl:sort select="sName" order="descending"
           case-order="lower-first" />
```

All of the attributes are optional. If no attribute values are specified then the nodes are sorted in ascending order with uppercase characters sorted first and the language determined by the operating system.

The `sort` instruction is always used within an `<xsl:for-each>` or `<xsl:apply-templates>` tag. The following code adds the `sort` instruction to a `for-each` instruction so that the values from the portfolio/stock node set are sorted in descending order of the `sName` element:

```
<xsl:for-each select="portfolio/stock">
    <xsl:sort select="sName" order="descending" />
    <xsl:value-of select="sName" />
</xsl:for-each>
```

When sorting is applied to a template, the `<xsl:apply-templates>` tag is entered as a two-sided tag as demonstrated in the following code:

```
<xsl:apply-templates select="portfolio/stock">
    <xsl:sort select="sName" order="descending" />
</xsl:apply-templates>
```

If you don't include a `select` attribute with the `sort` instruction, the XSLT processor will assume that you want to sort based on the value of the context node. Thus, the following code sorts the portfolio/stock/`sName` node set in descending order of the `sName` element:

```
<xsl:for-each select="portfolio/stock/sName">
    <xsl:sort order="descending" />
</xsl:for-each>
```

If you need to sort by more than one factor, you nest the `sort` instructions. The following code applies a template for the portfolio/stock node set, sorting the results first by category and then by `sName` within category:

```
<xsl:apply-templates select="portfolio/stock">
    <xsl:sort select="category" />
    <xsl:sort select="sName" />
</xsl:apply-templates>
```

### TIP

Always set the data type to number when sorting numeric values; otherwise, the numeric values will be sorted in alphabetical order with a value such as 123 listed before 24.

### REFERENCE

#### Sorting a Node Set

- To sort a node set, apply

```
<xsl:sort select="node set" data-type="text|number|qname"
           order="ascending|descending"
           case-order="upper-first|lower-first"
           lang="language" />
```

where `node set` is an XPath expression, the `data-type` attribute specifies the type of data to be sorted, the `order` attribute defines whether to sort in ascending or descending order, the `case-order` attribute specifies whether uppercase or lowercase characters are to be sorted first, and the `lang` attribute defines the language used to determine sort order.

The `sort` instruction can only be used within the `<xsl:for-each>` tag or the `<xsl:apply-templates>` tag.

Rafael has also noticed that the report lists the stocks in the same order in which they appear in the stock.xml file. He wants to see stocks reported alphabetically by stock name. You make this change now by applying the sort instruction to the `<xsl:apply-templates>` tag.

### To sort the stocks by order of name:

- 1. Return to the **stock.xsl** file in your text editor.
- 2. Go to the root template near the top of the style sheet and replace the `apply-templates` instruction with the following code:

```
<xsl:apply-templates select="portfolio/stock">
    <xsl:sort select="sName" />
</xsl:apply-templates>
```

Figure 5-32 highlights the modified code to sort the applied templates by the `sName` element.

**Figure 5-32 Applying a template in sorted order**

applies the template for the portfolio/stock node set sorted by the value of the `sName` element

```
<h1>Chesterton Financial</h1>
<h2>Portfolio Stocks</h2>

</header>
<section>
    <xsl:apply-templates select="portfolio/stock">
        <xsl:sort select="sName" />
    </xsl:apply-templates>
</section>
</body>
</html>

</xsl:template>
```

- 3. Save your changes to the style sheet and then regenerate the result document in your web browser or XML editor.
- 4. Verify that the stocks are now listed in ascending order of name with the Aluminum Company of America listed first, followed by the American Electric Power Company, and so forth.

**Trouble?** Make sure that you change the `<xsl:apply-templates>` tag from an empty tag to a two-sided tag by removing the `(/)` character at the end of the opening tag.

Rafael wants to know whether a particular stock is increasing or decreasing in value from its opening price. While this information can be gleaned from examining the Current Stock Values table, he would like the style sheet to provide this information graphically. This can be accomplished using conditional processing.

## Conditional Processing

Conditional processing is a programming technique that applies different styles based on the values from the source document. For example, the style sheet would write one result if a node from the source document has one particular value and a different

result if that node has a different value. One way of accomplishing this is with the following `if` instruction:

```
<xsl:if test="expression">
    styles
</xsl:if>
```

where `expression` is an XPath expression that is either true or false and `styles` are XSLT styles that are applied if the expression is true. To create an expression that returns a true or false value, you need to work with comparison operators.

## Using Comparison Operators and Functions

XPath supports several different **comparison operators** to compare one value to another. The most commonly used comparison operator is the equals symbol (=), which is used to test whether the two values are equal. For example, the following `if` statement tests whether the symbol attribute of the `sName` element is equal to "BA":

```
if test="sName/@symbol='BA'"
```

Note that when you nest a text string within the test attribute you need to use single quotes for the text string and double quotes for the test attribute or double quotes for the test string and single quotes for the test attribute.

Figure 5-33 describes other comparison operators supported by the test attribute used to compare text strings and numeric values.

**Figure 5-33** Comparison operators in XSLT

Operator	Description	Example
=	Tests whether two values are equal to each other	@symbol = "BA"
!=	Tests whether two values are unequal	@symbol != "BA"
&lt;	Tests whether one value is less than another	day &lt; 5
&lt;=	Tests whether one value is less than or equal to another	day &lt;= 5
>	Tests whether one value is greater than another	day > 1
&gt;=	Tests whether one value is greater than or equal to another	day >= 1
and	Combines two expressions, returning a value of true only if both expressions are true	@symbol = "BA" and day > 1
or	Combines two expressions, returning a value of true if either expression is true	@symbol = "BA" or @symbol = "UCL"
not	Negates the value of the expression, changing true to false or false to true	not(day >= 1)

### TIP

Be careful when comparing node sets and single values. When multiple values are involved, the expression is true if any of the values in the node set satisfy the test condition.

Because XSLT treats the left angle bracket character (<) as the opening character for an element tag, you must use the text string &lt; when you want to make less-than comparisons. XSLT doesn't have a problem with the right angle bracket character (>), however. As a result, one way to avoid using the &lt; expression is to reverse the order of a comparison. For example, instead of writing a comparison as

```
day &lt; 5
```

to test whether the value of the `day` element is less than 5, you write it as

```
5 > day
```

which gives the equivalent result, testing whether 5 is greater than the value of the day element. Comparison tests can be combined using the `and` and `or` operators. For example, the expression

```
day > 2 and day < 5
```

tests whether the value of the day element lies between 2 and 5. Similarly, the expression

```
@symbol = "BA" or @symbol = "AEP"
```

tests whether the value of the symbol attribute is equal to "BA" or "AEP". You can reverse the `true/false` value of an expression using the `not()` function. The expression

```
not(@symbol = "BA")
```

returns a value of `false` if the value of the symbol attribute is equal to "BA" and `true` if the symbol attribute is not equal to "BA".

## REFERENCE

### Testing a Condition with the `if` Element

- To test for a single condition, use the following `if` instruction:

```
<xsl:if test="expression">
    styles
</xsl:if>
```

where `expression` is an XPath expression that is either true or false and `styles` are XSLT commands that are run if the expression is true.

You can also test for the presence or absence of an attribute or element by entering the attribute or element name without any comparison operator. The following code tests for the presence of the symbol attribute. If the context node has that attribute, then the attribute's value is displayed; otherwise, nothing is done.

```
<xsl:if test="@symbol">
    <xsl:value-of select="@symbol" />
</xsl:if>
```

Verifying the presence or absence of an element or attribute is often used to avoid errors that would cause a transformation to fail. For example, if the style sheet had attempted to return the value of a missing symbol attribute, the processor would have halted the transformation and returned an error message. Using a comparison test avoids this problem.

## Testing for Multiple Conditions

Unlike other programming languages, XSLT doesn't support an `else-if` construction. This means that the `if` instruction tests for only one condition and allows for only one outcome. If you want to test for multiple conditions and display different outcomes, you need to apply the following `choose` structure:

```
<xsl:choose>
    <xsl:when test="expression1">
        styles
    </xsl:when>
    <xsl:when test="expression2">
        styles
    </xsl:when>
    ...

```

```
<xsl:otherwise>
    styles
</xsl:otherwise>
</xsl:choose>
```

where *expression1*, *expression2*, and so forth are expressions that are either true or false. The XSLT processor proceeds through the list of expressions one at a time and, when it encounters an expression that is true, it processes the corresponding style and stops. If it reaches the last expression without finding one that is true, the style contained in the `otherwise` instruction is processed.

## REFERENCE

### Testing Multiple Conditions with the choose Element

- To test multiple conditions, use the following `choose` element:

```
<xsl:choose>
    <xsl:when test="expression1">
        styles
    </xsl:when>
    <xsl:when test="expression2">
        styles
    </xsl:when>
    ...
    <xsl:otherwise>
        styles
    </xsl:otherwise>
</xsl:choose>
```

where *expression1*, *expression2*, and so forth are expressions that are either true or false.

Rafael wants to use the `choose` structure to test whether the stock's value is rising from its opening value. In other words, he wants to test whether the value of the current attribute of the `today` element is less than, greater than, or equal to the value of the `open` attribute. Depending on the result of the test, he wants the table of current stock values to display either a down-arrow image, an up-arrow image, or an even level image. He has three graphic files named `down.png`, `up.png`, and `same.png` for this purpose. Thus, to display different image files based on the change in the stock price, you'll employ the following `choose` structure:

```
<xsl:choose>
    <xsl:when test="@current < @open">
        
    </xsl:when>
    <xsl:when test="@current > @open">
        
    </xsl:when>
    <xsl:otherwise>
        
    </xsl:otherwise>
</xsl:choose>
```

You add this `choose` structure to the style sheet, creating a table cell that displays different icons based on the comparison of the stock's current value to its opening value.

### To apply the choose element:

- 1. Return to the **stock.xsl** file in your editor and go to the today template.
- 2. Within the **<th>** tag for the Current Stock Values, change the value of the **colspan** attribute to **6** because you'll be adding a sixth cell to the table row.
- 3. Insert a new table heading cell in the second row of the table by adding the tag **<th></th>**, which creates a blank table cell to the left of the Current column heading.
- 4. Scroll down to the table body and insert the following code for a new table cell displaying one of three possible icon images:

```

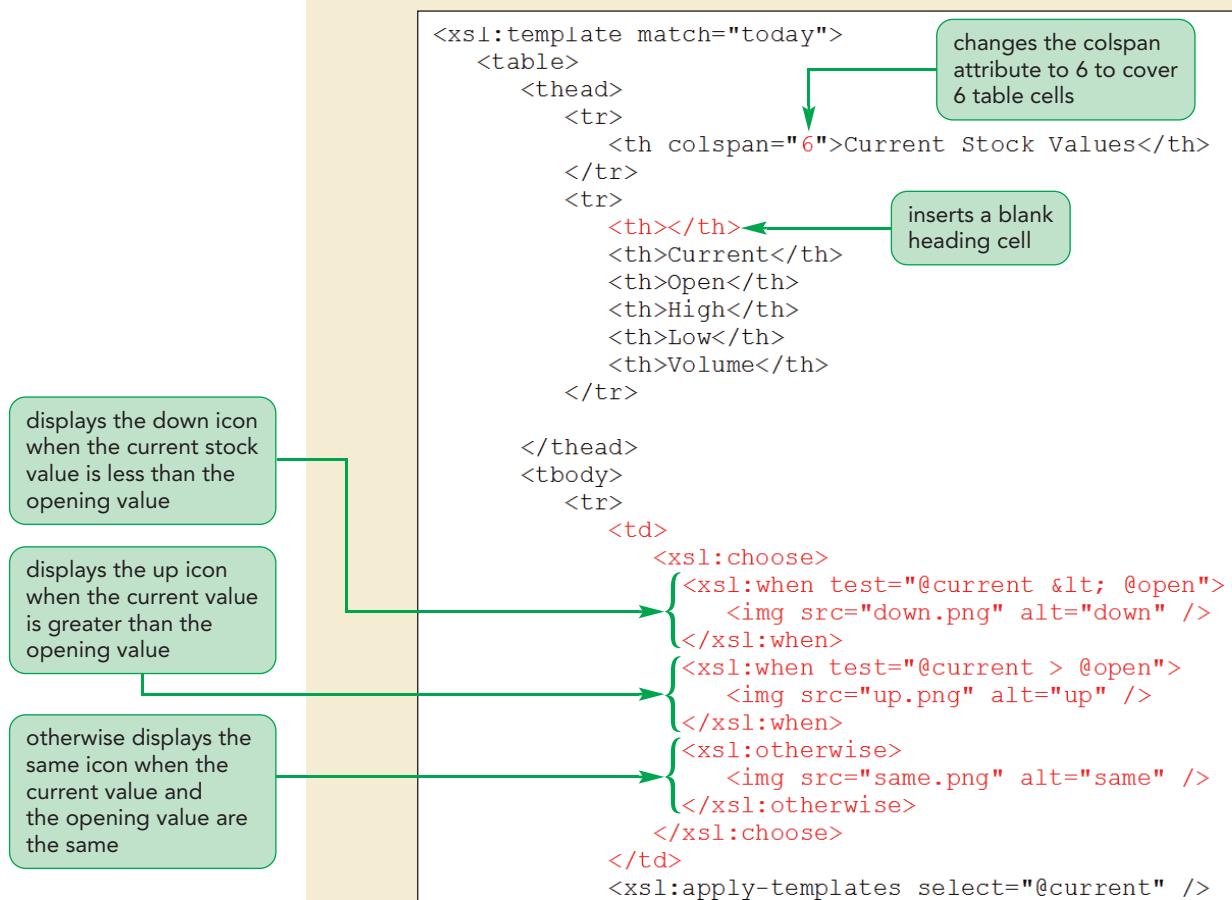
<td>
  <xsl:choose>
    <xsl:when test="@current < @open">
      
    </xsl:when>
    <xsl:when test="@current > @open">
      
    </xsl:when>
    <xsl:otherwise>
      
    </xsl:otherwise>
  </xsl:choose>
</td>

```

Figure 5-34 shows the revised code in the style sheet.

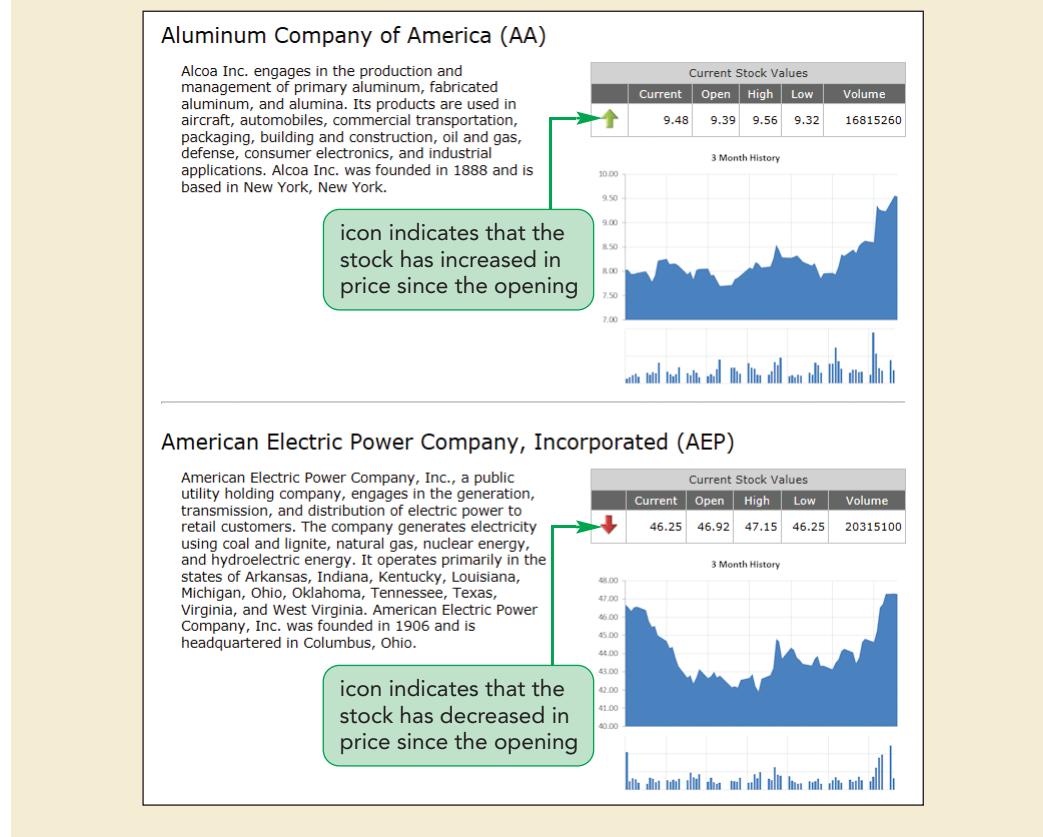
Figure 5-34

### Applying the choose structure to display different inline images



5. Save your changes to the file and then regenerate the result document. Figure 5-35 shows the newly added icon for the first two stocks listed in the web page.

**Figure 5-35** Icons representing the change in stock price



The stocks in the stock.xml file are placed into three categories—Industrials, Utilities, and Transportation. Rafael wants the result document to group the stocks based on those categories. One way to accomplish this is to use predicates.

## Filtering XML with Predicates

A predicate is part of a location path that restricts the node set to only those nodes that fulfill a specified condition. The general syntax for a predicate is

*node-set*[*condition*]

where *node-set* is an XPath expression that references a particular node set and *condition* is an expression for a condition that any nodes in the node set must fulfill. The expression in the predicate can use the same conditional operators used with the *if* and *choose* instructions. For example, the predicate

```
sName[ @symbol = "BA" or @symbol="AEP" ]
```

matches all *sName* elements whose symbol attribute is equal to either “BA” or “AEP”. If you don’t include a value for an attribute, the expression selects only those nodes that contain the attribute. For example, the expression

```
sName[ @symbol ]
```

selects only those *sName* elements that have a *symbol* attribute.

## Predicates and Node Position

A predicate can also indicate the position of a node in the node tree. The general syntax is

```
node-set[position]
```

where *position* is an integer indicating the position of the node. For example, the expression

```
stock[3]
```

selects the third stock element from the source document. The union operator ( | ) can also be used to select multiple positions. Thus, the expression

```
stock[3|5]
```

selects the third and fifth stock elements.

## Predicates and Functions

A predicate can also contain an XPath function. The two functions that you'll explore in this tutorial are the `last()` and `position()` functions. The `last()` function returns the last node in the node tree. Thus, the expression

```
stock[last()]
```

returns the last stock element from the source document's node tree. The `position()` function returns the position value of the node. For example, the following expression selects the second stock element:

```
stock[position()=2]
```

and is equivalent to the expression `stock[2]`. The `position()` function is useful when combined with comparison operators to select ranges of nodes. The following expression combines the `position()` function with comparison operators to select the second through fifth stock nodes:

```
stock[position()>=2 and position()<=5]
```

Finally, predicates can be used within longer location paths. For example, the following code returns the value of the `sName` element but only for the second stock listed in the source document:

```
<xsl:value-of select="stock[position()=2]/sName" />
```

## Using Node Predicates

- To select a subset of nodes from the node tree, combine the node set with a predicate using the syntax

*node-set[condition]*

where *node-set* is an XPath expression that references a particular node set and *condition* is an expression for a condition that nodes in the node set must fulfill.

- To process only the first node from a branch of the node tree, use the expression

*node-set[1]*

- To process only the *n*th node from a branch of the node tree, use the expression

*node-set[n]*

- To process only the last node from a branch of the node tree, use the expression

*node-set[last()]*

- To process a node from a specific location in the node's tree branch, use the expression

*node-set[position()=value]*

where *value* is an integer indicating the node's location in the branch.

Rafael wants to display the name of each stock category using an h1 heading. Under each heading, Rafael wants to display the stocks that belong to that category (Industrials, Utilities, Transportation) sorted in alphabetical order by stock name. The code to display the industrial stocks uses the following predicate with the *apply-templates* element:

```
<h1>Industrials</h1>
<xsl:apply-templates
    select="portfolio/stock[category='Industrials']">
    <xsl:sort select="sName" />
</xsl:apply-templates>
```

The code to display stocks that belong to the Utilities and Transportation categories is similar. You add this code to the root template of the stock.xsl style sheet.

### To apply a predicate to the selection of stock nodes:

- 1. Return to the **stock.xsl** file in your text editor and go to the root template, replacing the contents within the second section element (the one immediately following the *</header>* tag) with the following:

```
<h1>Industrials</h1>
<xsl:apply-templates
    select="portfolio/stock[category='Industrials']">
    <xsl:sort select="sName" />
</xsl:apply-templates>
```

- 2. Copy the section element you just created including the opening and closing section tags and paste the code as a new section under the section that contains the code for the Industrials head, replacing Industrials with **Utilities** in both the h1 heading and the predicate category.

- 3. Paste a third section under the section you just created for the Utilities with **Transportation** for the h1 heading and the predicate category.

Figure 5-36 highlights the new and revised code in the style sheet.

**Figure 5-36**

### Limits node sets using a predicate

```

</header>
<section>
  <h1>Industrials</h1>
  <xsl:apply-templates
    select="portfolio/stock[category='Industrials']">
    <xsl:sort select="sName" />
  </xsl:apply-templates>
</section>

<section>
  <h1>Utilities</h1>
  <xsl:apply-templates
    select="portfolio/stock[category='Utilities']">
    <xsl:sort select="sName" />
  </xsl:apply-templates>
</section>

<section>
  <h1>Transportation</h1>
  <xsl:apply-templates
    select="portfolio/stock[category='Transportation']">
    <xsl:sort select="sName" />
  </xsl:apply-templates>
</section>

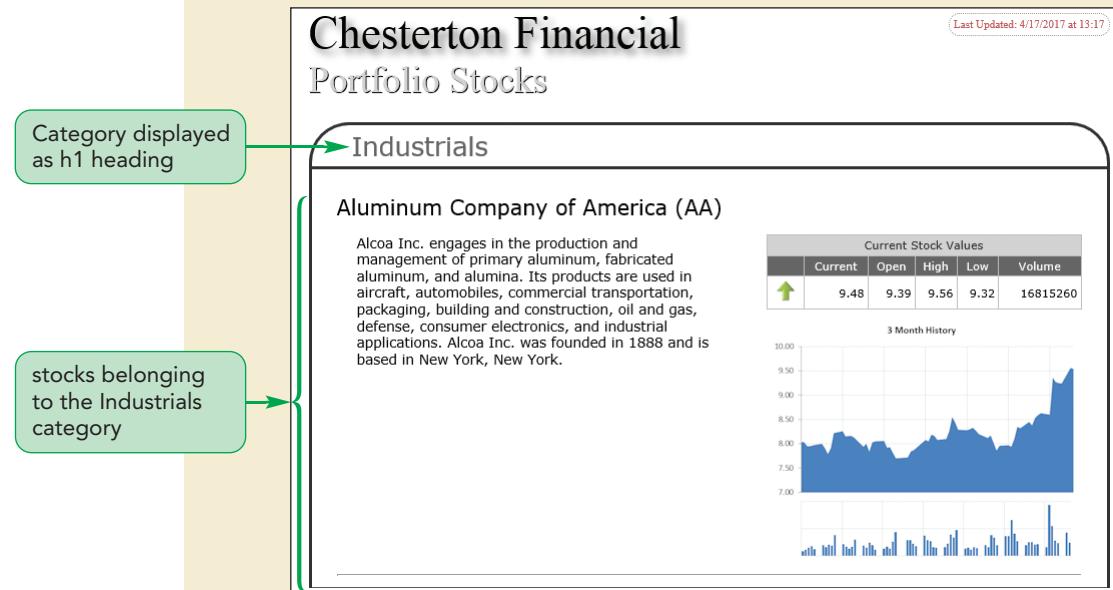
</body>
</html>

```

- 4. Save your changes.
- 5. Use your web browser or XML editor to regenerate the result document. Verify that the stocks are now grouped into categories and that, within each category, stocks are listed alphabetically (see Figure 5-37, which shows the first stock in the Industrials category).

Figure 5-37

Final appearance of the Web page



- 6. Close your web browser or XML editor, and then close the **stock.xsl** file.

In addition to categorizing nodes, you can use predicates for many other applications. In future tutorials you'll examine ways to write predicates to create more advanced and dynamic node structures.



## Written Communication: Writing Effective XSLT Style Sheets

XSLT style sheets need to be written clearly both to make the code easier for your colleagues to read and interpret and to minimize the chance of coding mistakes. Here are some tips to keep in mind as you develop your own style sheets.

- Build up your style sheet incrementally by adding only a few templates at a time. Verify that the style sheet works at each stage before introducing additional complexity.
- Add comments to your style sheet document to aid yourself and others in interpreting and revising the style sheet at a later time.
- Use templates in preference to the `<xsl:for-each />` tag whenever possible. Templates are faster to process, easier to debug, and can be reused in different contexts throughout the style sheet.
- Use relative path expressions within templates and for-each loops. Any XPath expression should be interpreted relative to the context node and not rely on nodes defined outside of the template or for-each loop.
- Reduce errors and speed up processing by using the `indent="no"` attribute with the `<xsl:output />` tag. This will remove extra white space and reduce the size of the result document. Note that the default value for HTML files is `indent="yes"`, while for XML documents the default is `indent="no"`.
- Use an XML editor and avoid coding by hand. This will cut down on typos and syntax errors.
- Use the `<xsl:comment />` tag to add comments to the result document, making it easier for others to interpret the code in that file.
- The web contains a wealth of free XSLT libraries. Before trying to solve a coding problem, check these libraries to see if a solution is already available.

A well-written style sheet will also be processed quicker, which is an important consideration when transforming large files containing thousands of records.

## Constructing Elements and Attributes with XSLT

In writing the code for the stock report, you entered the HTML tags for the elements and attributes that were part of the final result document. In the process of creating the HTML file, the XSLT processor used these HTML tags to create a **result tree**, which is composed of the element, attribute, text, and other nodes.

For some projects, you might need to have a result tree whose nodes are defined explicitly based on contents of the source document. This happens most often when changing the structure of one XML document into another. For example, the source document might contain the following elements and attributes:

```
<stock>
  <sName symbol="BA">Boeing Company</sName>
  <values>
    <day>124.81</day>
    <day>125.22</day>
  </values>
</stock>
```

which you want to restructure as

```
<stock>
  <stockName>Boeing Company</stockName>
  <stockSymbol>BA</stockSymbol>
  <stockValues day1="124.81" day2="125.22" />
</stock>
```

To create this document you can have XSLT construct the element and attribute nodes, rather than entering them as literal result elements.

## Constructing an Element Node

To construct an element node in the result tree, XSLT uses the following `<xsl:element>` tag:

```
<xsl:element name="text" namespace="uri">
    styles
</xsl:element>
```

where the `name` attribute assigns a name to the element and the `namespace` attribute provides a namespace. For example, to convert the `sName` element and `symbol` attribute into elements named `stockName` and `stockSymbol` you could apply the following code:

```
<xsl:element name="stockName">
    <xsl:value-of select="sName" />
</xsl:element>
<xsl:element name="stockSymbol">
    <xsl:value-of select="sName/@symbol" />
</xsl:element>
```

Under this transformation, the source document code

```
<sName symbol="BA">Boeing Company</sName>
```

is transformed into

```
<stockName>Boeing Company</stockName>
<stockSymbol>BA</stockSymbol>
```

### TIP

If you need to create a one-sided element, you use a one-sided `<xsl:element />` tag.

New elements can be combined with predicates, conditional statements, and the `for-each` instruction to create a subset of the original source document in a new structure. For example, to create an XML document containing a list of the stock symbols for transportation stocks, you could apply the following template:

```
<xsl:template match="portfolio">
    <xsl:element name="Transportation">
        <xsl:for-each select="stock[category='Transportation']">
            <xsl:element name="stockSymbol">
                <xsl:value-of select="sName/@symbol" />
            </xsl:element>
        </xsl:for-each>
    </xsl:element>
</xsl:template>
```

After transforming the `stock.xml` document, an XSLT processor generates the following `Transportation` element containing symbols of all of the transportation stocks:

```
<Transportation>
    <stockSymbol>CNI</stockSymbol>
    <stockSymbol>DAL</stockSymbol>
    <stockSymbol>R</stockSymbol>
    <stockSymbol>LUV</stockSymbol>
    <stockSymbol>UNP</stockSymbol>
</Transportation>
```

## Constructing Attributes and Attribute Sets

Attributes are constructed in XSLT using the following `<xsl:attribute />` tag:

```
<xsl:attribute name="text" namespace="uri">
    styles
</xsl:attribute>
```

where the `name` attribute specifies the name of the attribute and the `namespace` attribute indicates the namespace. Attributes are nested within `<xsl:element>` tags so that the attribute is added to the newly created element. For example, to convert this collection of nodes

```
<values>
    <day>124.81</day>
    <day>125.22</day>
</values>
```

into a single element node with the following attributes:

```
<stockValues day1="124.81" day2="125.22" />
```

you can apply the following transformation:

```
<xsl:element name="stockValues">
    <xsl:attribute name="day1">
        <xsl:value-of select="day[1]" />
    </xsl:attribute>
    <xsl:attribute name="day2">
        <xsl:value-of select="day[2]" />
    </xsl:attribute>
</xsl:element>
```

### TIP

The advantage of attribute sets is that a collection of common attributes can be easily applied to several different elements.

Rather than nesting the entire collection of attributes, those attributes can be grouped within an **attribute set**, which allows you to add several attributes to the same element without having a long nested statement. To create an attribute set you apply the following `attribute-set` element:

```
<xsl:attribute-set name="text" use-attribute-sets="name-list">
    <xsl:attribute name="text">styles</xsl:attribute>
    <xsl:attribute name="text">styles</xsl:attribute>
    ...
</xsl:attribute-set>
```

where the `name` attribute contains the name of the set and then the names of the individual attributes created within that set. You can also refer to other attribute sets by specifying their names in the `name-list` parameter, allowing you to build a collection of attribute sets by combining one with another. For example, the following code creates an attribute set named `dayAttributes`, which consists of values for the `day` element:

```
<xsl:attribute-set name="dayAttributes">
    <xsl:attribute name="day1">
        <xsl:value-of select="day[1]" />
    <xsl:attribute>
    <xsl:attribute name="day2">
        <xsl:value-of select="day[2]" />
    <xsl:attribute>
</xsl:attribute>
```

The attribute set is then applied to an element by applying the `attribute-set` attribute. The following code demonstrates how to apply the `dayAttributes` attribute-set to the `stockValues` element:

```
<xsl:element name="stockValues" attribute-set="dayAttributes" />
```

**REFERENCE**

### *Creating Elements and Attributes with XSLT*

- To create an element, use the XSLT element

```
<xsl:element name="text" namespace="uri"
    use-attribute-sets="namelist">
    styles
</xsl:element>
```

where the `name` attribute assigns a name to the element, the `namespace` attribute provides a namespace, and `use-attribute-sets` provides a list of attribute sets.

- To create a one-sided or empty element, use

```
<xsl:element attributes />
```

- To create an attribute, use

```
<xsl:attribute name="text" namespace="uri">
    styles
</xsl:attribute>
```

where the `name` attribute specifies the name of the attribute and the `namespace` attribute indicates the namespace.

- To create a set of attributes, use

```
<xsl:attribute-set name="text" use-attribute-sets="name-list">
    <xsl:attribute name="text">styles</xsl:attribute>
    <xsl:attribute name="text">styles</xsl:attribute>
    ...
</xsl:attribute-set>
```

where the `name` attribute is the name of the set and `use-attribute-sets` can refer to the contents of another attribute set.

## Constructing Comments and Processing Instructions

XSLT also includes elements to write comments and processing instructions to the result tree. To construct a comment node, use the element

```
<xsl:comment>
    comment text
</xsl:comment>
```

where `comment text` is the text that should be placed within a comment tag. For example, the code

```
<xsl:comment>
    Sample Stock Portfolio
</xsl:comment>
```

creates the following comment in the result document:

```
<!-- Sample Stock Portfolio -->
```

To create a processing instruction node, use the element

```
<xsl:processing-instruction name="text">  
    attributes  
</xsl:processing-instruction>
```

where the `name` attribute provides the name of the processing instruction and `attributes` are attributes contained within the processing instruction. For example, if you want to add a processing instruction to attach the `styles.css` style sheet to the result document, you use the code

```
<xsl:processing-instruction name="xmlstylesheet">  
    href="styles.css" type="text/css"  
</xsl:processing-instruction>
```

which generates the following tag in the result document:

```
<?xml-stylesheet href="styles.css" type="text.css"?>
```

The `processing-instruction` element contains the attributes of the processing instruction to be placed in the result document.

## REFERENCE

### *Creating Comments and Processing Instructions*

- To add a comment to the result document, use

```
<xsl:comment>  
    comment text  
</xsl:comment>
```

where `comment text` is the text to be placed in the comment.

- To add a processing instruction to the result document, use

```
<xsl:processing-instruction name="text">  
    attributes  
</xsl:processing-instruction>
```

where the `name` attribute provides the name of the processing instruction and `attributes` are attributes and attribute values contained in the processing instruction.

You don't have to use XSLT to create elements, attributes, comments, or processing instructions in Rafael's stock report, but you keep these features of XSLT in mind for future projects. For now, you've completed your work on the stock report. Rafael will examine the document and discuss the results with his colleagues. If he needs to make changes, he'll contact you.

**REVIEW****Session 5.3 Quick Check**

1. An XML document contains the root element named books, which has a single child element named book. Each book element has two child elements named title and author. Using the XSLT's for-each instruction, sort the book elements in alphabetical order based on the title.
2. By default, does the `<xsl:sort>` element sort items numerically or alphabetically?
3. The book element has a single attribute named category. The value of category can be either fiction or nonfiction. Write an if construction that displays the book title only if it is a nonfiction book.
4. Use a choose construction that displays book titles in an h3 heading if they are fiction and an h2 heading if otherwise.
5. Correct the following expression so that it doesn't result in an error:

```
test="sales < 20000"
```

6. What code do you use to select the first book from the XML document described in Question 1? What do you use to select the last book?
7. What code do you use to create an element named inventory that contains the value 15000?
8. What code do you use to create an empty element named inventory that contains a single attribute named amount with the value 15000?

**PRACTICE**

## Review Assignments

**Data Files needed for the Review Assignments:** `portfoliotxt.xml`, `portfoliotxt.xsl`, +1 CSS file, +3 PNG files

Rafael Garcia has worked with the stock report you generated and has made a few modifications to the source document. Rafael has added a few more elements providing information on each stock's high and low value over the previous year, the stock's P/E ratio, earnings per share (EPS), dividend, and yield. He's also included the URL to the company's website.

Rafael wants to try a new design, in which these new stock values appear in a table alongside a description of the stock. He also wants to show the current stock value and whether it's rising, falling, or remaining level, which is to be prominently displayed below the company name. Figure 5-38 shows a preview of the revised content and layout of the report.

**Figure 5-38** Revised stock report design

The screenshot displays a web page titled "Chesterton Financial Portfolio Stocks". At the top right, there is a small red box containing the text "Last Updated: 4/17/2017 at 13:17". Below the title, the section "Industrials" is shown. Under this section, two stocks are listed: "Aluminum Company of America (AA)" and "ATT Corporation (T)".

**Aluminum Company of America (AA)**  
**↑9.48**  
 Alcoa Inc. engages in the production and management of primary aluminum, fabricated aluminum, and alumina. Its products are used in aircraft, automobiles, commercial transportation, packaging, building and construction, oil and gas, defense, consumer electronics, and industrial applications. Alcoa Inc. was founded in 1888 and is based in New York, New York.

**ATT Corporation (T)**  
**↓35.78**  
 ATT Inc. provides telecommunications services to consumers, businesses, and other providers in the United States and internationally. The company operates in three segments: Wireless, Wireline, and Other. The Wireless segment offers various wireless voice and data communication services. The Wireline segment provides data services and network integration. The Other segment provides application management, security services, and satellite video services. The company was founded in 1983 and is based in Dallas, Texas.

Rafael has already created the CSS style sheet for this web page, but he needs you to create the XSLT style sheet, which will generate the HTML code used in the report.

Because creating a style sheet can be complicated, it is strongly recommended that you save your changes and generate the result document as you complete each step below to check on your progress and detect any problems early. Complete the following:

1. Using your text editor, open the `portfoliotxt.xml` and `portfoliotxt.xsl` files from the `xml05` review folder. Enter **your name** and the **date** in the comment section of each file, and save them as `portfolio.xml` and `portfolio.xsl`, respectively.
2. Go to the `portfolio.xml` file in your text editor. Take some time to review the content of the file and its structure. Add a processing instruction after the comment section that attaches the `portfolio.xsl` style sheet to this XML document. Close the file, saving your changes.
3. Go to the `portfolio.xsl` file in your text editor. Below the comment section, set up this document as an XSLT style sheet by adding a stylesheet root element and declaring the XSLT namespace using the namespace prefix `xsl`.
4. Add an `output` element to indicate to the XSLT processor that the transformed file should be in HTML5 format.

5. Create a root template and, within that template, enter the following HTML code:

```
<html>
  <head>
    <title>Portfolio Stocks</title>
    <link href="portfolio.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
  </body>
</html>
```

6. Directly after the opening `<body>` tag in the root template, insert the following code:

```
<header>
  <section>
    Last Updated: date at time
  </section>
  <h1>Chesterton Financial</h1>
  <h2>Portfolio Stocks</h2>
</header>
```

where *date* and *time* are the values of the date and time elements from the source document using the XSLT `value-of` elements.

7. Directly below the closing `</header>` tag, insert the following HTML code three times, one for each of three categories of stock (Industrials, Utilities, and Transportation):

```
<section>
  <h1>Category</h1>
  stock template
</section>
```

where *category* is either Industrials, Utilities, or Transportation, and *stock template* applies the template for elements from the portfolio/stock location path for stocks of the specified category. Sort the applied templates in alphabetical order by stock name.

8. Create a template for the today element, and then, within the template, enter the following HTML code:

```
<table>
  <tr>
    <th>Current</th>
    <th>Open</th>
    <th>High</th>
    <th>Low</th>
    <th>Volume</th>
  </tr>
  <tr>
    <td>current</td>
    <td>open</td>
    <td>high</td>
    <td>low</td>
    <td>volume</td>
  </tr>
</table>
```

where *current*, *open*, *high*, *low*, and *volume* are the values of the current, open, high, low, and vol attributes using the XSLT `value-of` element.

9. Create a template for the summary element, and then, within the template, enter the following HTML code:

```
<table>
  <tr>
    <th>Web Address</th>
    <td><a href="link">link</a></td>
  </tr>
```

```

<tr>
    <th>52wk. Range</th>
    <td>low - high</td>
</tr>
<tr>
    <th>P/E</th>
    <td>pe_ratio</td>
</tr>
<tr>
    <th>EPS</th>
    <td>eps</td>
</tr>
<tr>
    <th>Div. and Yield</th>
    <td>dividend (yield)</td>
</tr>
</table>

```

where *link*, *low*, *high*, *pe\_ratio*, *eps*, *dividend*, and *yield* are the values of the corresponding elements from the source document using the XSLT `value-of` element.

10. Create a template for the stock element and, within the template, enter the following HTML code:

```

<article>
    today
    summary
    <h1>stock name (symbol)</h1>
</article>

```

where *today* and *summary* apply the today and summary templates, *stock name* is the name of the stock, and *symbol* is the value of the symbol attribute using the XSLT `value-of` element.

11. Directly between the closing `</h1>` and closing `</article>` tags in the stock template, use a choose structure to insert the following h2 heading and paragraph:

```

<h2 class="change">
    
    current
</h2>
<p>description</p>

```

where *change* has the value “up”, “down”, or “same” depending on whether the value of the current attribute is less than, greater than, or equal to the open attribute; *current* is the value of the current attribute; and *description* is the value of the description element. Note that the current and open attributes must be referenced from within the today element. Be sure to replace the HTML code with the proper value of expressions.

12. Save your changes to the portfolio.xml file.
13. Generate your result document using either an XML editor or your web browser. Verify that the layout matches what is shown in Figure 5-38 and that each link in the summary table jumps the user to the corresponding stock’s website.

## APPLY

### Case Problem 1

**Data Files needed for this Case Problem:** itemstxt.xml, librarytxt.xsl, +1 CSS file, +2 PNG files

**Denison Public Library** Helen Young is a programmer at Denison Public Library in Ennis, Montana. The library has recently set up a database system in which lists of library items are written to XML documents. Helen wants to be able to view those lists as HTML code within her web browser. She has asked you to help develop an XSLT style sheet to write that HTML code. A preview of the web page is shown in Figure 5-39.

Figure 5-39 Library item list

Location	Collection	Call No.	Status
Adult	DVDs	DVD MAN	Available

Location	Collection	Call No.	Status
Adult	Non-fiction	511.2 GAR	Checked Out (due: 3/23/2017)

Location	Collection	Call No.	Status
Adult	DVDs	DVD ALI	Available

Location	Collection	Call No.	Status
Adult	Non-fiction	973.91 SHL	Checked Out (due: 3/22/2017)

The sample list she's compiled includes books and DVDs. Her proposed web page uses icons to differentiate books from visual materials. If an item is currently checked out, she wants the page to display the due date for the item. The list should be sorted alphabetically by the item name. Helen has already created a CSS style sheet for the page; your job will be to create the XSLT style sheet. Complete the following:

1. Using your text editor, open the **itemstxt.xml** and **librarytxt.xsl** files from the **xml05 ▶ case1** folder. Enter **your name** and the **date** in the comment section of each file, and save them as **items.xml** and **library.xsl**, respectively.
2. Go to the **items.xml** file in your text editor. Review the content of the file and its structure. Add a processing instruction after the comment section that attaches the **library.xsl** style sheet to this XML document. Close the file, saving your changes.
3. Go to the **library.xsl** file in your text editor and go down to the root template. Add the following commands to the template:

```

<html>
  <head>
    <title>Denison Public Library</title>
    <link href="libstyles.css"
      rel="stylesheet" type="text/css" />
  </head>
  <body>
    <header>
      <h1>Denison Public Library</h1>
      <h2>Ennis, Montana</h2>
    </header>
  </body>
</html>

```

4. Within the root template directly above the **</body>** tag, insert the following:

```

<section>
  <h1>Item List</h1>
  item template
</section>

```

where **item template** applies the template for the **itemlist/item** path, sorted by the **title** element.

5. Create a template for the item element to display information on each library item. Add the following HTML code to the template:

```
<article>
</article>
```

6. Within the <article></article> tags, insert the following to display the title and authors of the library item:

```
<h1>title</h1>
<h2>by: [author] [author] ...
```

where *title* is the value of the title element and [author] [author] ... is the list of authors for the work. (Hint: Use the **for-each** instruction to go through each author element in the authors/author path.)

7. Below the h2 heading, insert the following:

```
<p>
  Type:
  <span>
     type
  </span>
</p>
```

where *file* is either book or film depending on whether the value of the type element is “book” or “Visual Material” and *type* is the value of the type element. (Hint: Use a choose structure within the span element to choose whether to display the book.png image file or the film.png file.)

8. Below the closing </p> tag, insert the following:

```
<p>
  Tags:
  <span>
    subject/subject/.....
  </span>
</p>
```

where *subject/subject/...* is a list of the subject values within the subjects/subject path separated with the “/” symbol. (Hint: Use the **for-each** instruction with the subjects/subject path and display the value of the context node within the **for-each** element.)

9. Below the list of subjects, insert the following:

```
<p>
  Publisher:
  <span>publisher</span>
</p>
<p>
  ISBN:
  <span>isbn</span>
</p>
```

where *publisher* and *isbn* are the values of the publisher and isbn elements.

10. Below the closing </p> tag, insert the following table:

```
<table>
  <tr>
    <th>Location</th>
    <th>Collection</th>
    <th>Call No.</th>
    <th>Status</th>
  </tr>
  <tr>
    <td>location</td>
    <td>collection</td>
    <td>callno</td>
    <td>
      status
      (due: return)
    </td>
  </tr>
</table>
```

where *location*, *collection*, *callno*, and *status* are the values of the location, collection, callno, and status elements. If the status element contains a return attribute, display the text (due: *return*) where *return* is the value of the return attribute.

11. Save your changes to the file.
12. Generate your result document using either an XML editor or your web browser. Verify that the layout and content of each item matches that shown in Figure 5-39.

## CHALLENGE

### Case Problem 2

**Data Files needed for this Case Problem:** feedtxt.xsl, newstxt.xml, +1 CSS file, +5 HTML files

**Twin Life News Feed** Williams Lewis manages a monthly magazine and website for parents of twins called *Twin Life News*. A feature of the website is a newsfeed to highlight current articles of interest to the site's subscribers. RSS is an XML vocabulary designed for the reporting and distribution of current news features across the Internet. William wants you to create an XSLT style sheet to transform the content of the news feed into a format that is easily readable in a web browser.

Figure 5-40 shows a preview of the news feed as viewed within a browser.

**Figure 5-40** News feed Web page

**Twin Life News Feed**

You are viewing a feed that contains frequently updated content. Receive notifications on updates to this feed by subscribing to Twin Life News. Updated information from the feed is automatically downloaded to your computer and can be viewed in your browser.

[Subscribe](#)

**Multiple Births on the Increase**  
By: Miriam Lasker, PhD  
Tue, 07 Feb 2017 11:48:11 +0000

**Incidences of multiple birth pregnancies have increased by 8% in the past 10 years.** The increase has been attributed to later pregnancies by working mothers, which are more likely to result in twin births and the increased use of fertility drugs. "Twins and even triplet births are much more common these days," says nurse Denise Schoeder of St. Jude's Hospital in Beaverton, Illinois. "Fortunately we are much better equipped to handle the special complications that arise with these type of pregnancies than we were, even five years ago." Statistics bear this out. Complications resulting from low birthweight babies (a common occurrence with multiple births) are down 25% in the past ten years.

In Dade County, Florida, two quadruplets have been born in the past 18 months. "I think we'll need to increase the size of our maternity ward if this keeps up," says Dr. Lawrence Jacobs of Miami Community Hospital.

[More ...]

**Twins and Talking**  
By: Peter Kuhlman, PhD, Hughes Medical  
Thu, 26 Jan 2017 15:23:41 +0000

**Communication occurs long before speech.** Babies communicate with their parents through crying, laughter, smiling and pointing. The attentive parent quickly becomes attuned to this "non-speech" communication. At some point, usually in the second year of life, babies will begin to add a single words to these acts of expression. A baby will point to a cup and say, "drink." The names "Mama" and "Dada" are among the first out of the baby's mouth.

**Recent Articles**

- Dr. Miriam Lasker reports on the increase in multiple births
- Dr. Peter Kuhlman explores twin communication.
- Dr. Karen Kerman examines role playing in twins and looks at "role switching."
- Dr. Miriam Lasker reports on the events at the Midwest convention in Chicago.
- Dr. Lucas Lawson answers your questions on raising twins and multiples.

Each news item in the feed has a one- or two-paragraph outtake. The text of the outtake is written in HTML code embedded within a CDATA section and must be sent to the result document as text with no processing. Complete the following:

1. Using your text editor, open the **newstxt.xml** and **feedtxt.xsl** files from the **xml05 ▶ case2** folder. Enter **your name** and the **date** in the comment section of each file, and save them as **news.xml** and **feed.xsl**, respectively.
2. Go to the **news.xml** file in your text editor. Review the content of the file and its structure. Note the use of different namespaces for different RSS elements. Add a processing instruction after the comment section that attaches the **feed.xsl** style sheet to this XML document. Close the file, saving your changes.

3. Within the `stylesheet` element, insert the root template, writing the following code to the result document:

```
<html>
  <head>
    <title>title</title>
    <link href="feedstyles.css" rel="stylesheet"
      type="text/css" />
  </head>
  <body>
    </body>
</html>
```

where `title` is the value of the channel title from the `title` element in the `news.xml` file using the location path `/rss/channel/title`.

4. Directly below the `<body>` tag, insert the following code into the root template:

```
<div id="head">
  <h1>title</h1>
  <p>description</p>
  <p>
    <a href="link">Subscribe</a>
  </p>
</div>
```

where `title` is the channel title, `description` is the channel description, and `link` is the channel link.

5. Directly after the closing `</div>` tag, insert:

```
<div id="sidebar">
  <h1>Recent Articles</h1>
  <p>
    <a href="#guid">subhead</a>
  </p>
  <p>
    <a href="#guid">subhead</a>
  </p>
  ...
</div>
```

where `<p><a href="#guid">subhead</p></a>...` list each subhead of every item element in the document formatted as a paragraph with a hypertext link. Use the value of the `guid` element to identify the link. (Hint: Use the `for-each` instruction to go through the list of item elements.)

6. Directly below the `</div>` tag you added in the previous step, apply the template for the location path `/rss/channel/item`.

7. Create the template for the item element that will write the following code to the result document:

```
<div class="article">
  <h1 id="guid">title</h1>
  <h2>By: author</h2>
  <h3>pubDate</h3>
</div>
```

where `guid` is the value of the `guid` element, `title` is the item title, `author` is the item author, and `pubDate` is the item publication date.

-  **EXPLORE** 8. Between the closing `</h3>` and the closing `</div>` tags, insert the following code:

```
<div class="outtake">
  description
</div>
<p>[<a href="link">More ...</a>]</p>
```

where *description* contains the HTML code for the item description and *link* is the link for the news item. Disable escaping for the value of the *description* element so that the XSLT processor will submit the *description* value directly as text.

9. Save your changes to the file.
10. Generate your result document using either an XML editor or your web browser. Verify that the page layout and format resembles Figure 5-40. Also confirm that all of the recent article links listed in the box on the right margin point to the article headings and that by clicking the [More ...] link you can view a page containing the complete news article.

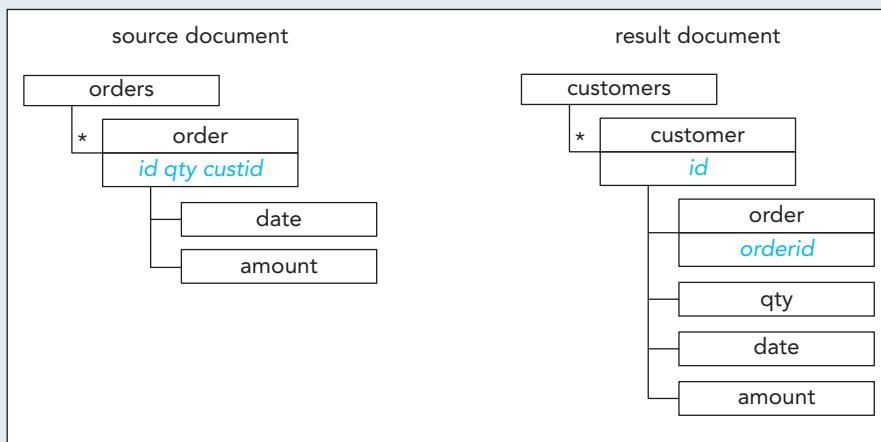
## CHALLENGE

### Case Problem 3

**Data Files needed for this Case Problem:** **clisttxt.xsl, orderstxt.xml, +1 DTD file**

**The Spice Bowl** Sharon Krueger manages accounts for *The Spice Bowl*, an online seller of spices in bulk quantities for restaurants and grocery stores. One of Sharon's colleagues has created an order report in an XML file detailing some of the orders placed at the Spice Bowl over the last few days. Rather than organizing the data by orders, Sharon would prefer to have an XML document that organizes the data by customer ID number. The layout of the file she was given and the file she would like to have are displayed in Figure 5-41.

**Figure 5-41 Structure of the source and result document**



She's asked you to write an XSLT style sheet to transform the layout of her file into the new structure. To create this style sheet, you'll work with the XSLT commands to create elements and attributes. Note that you must have access to an XML editor to generate the result document. Complete the following:

1. Using your text editor, open the **orderstxt.xml** and **clisttxt.xsl** files from the **xml05 ▶ case3** folder. Enter **your name** and the **date** in the comment section of each file, and save them as **orders.xml** and **clist.xsl**, respectively.
2. Go to the **orders.xml** file in your text editor. Add a processing instruction after the comment section that attaches the **clist.xsl** style sheet to this XML document. Close the file, saving your changes.
3. Go to the **clist.xsl** file in your text editor. Insert the root **stylesheet** element and namespace.
4. Add an **output** element to the style sheet that instructs the XSLT processor that the result document is an XML document, that the version is 1.0, that the text is indented, and that the XML document should contain a DOCTYPE system declaration pointing to the **customers.dtd** file.

-  **EXPLORE 5.** Insert the root template. Within the root template, create a comment node for the result document that contains the following text:

Author: *name*

Date: *date*

where *name* is your name and *date* is the current date.

-  **EXPLORE 6.** Within the root template create and add the customers element node to the result tree. Within this new element apply the order template using the pathname orders/order, sorted by the custid attribute.

-  **EXPLORE 7.** Create the order template. Within this template create the customer element node. Add the id attribute node to the customer element, using the custid attribute as the value.

8. Within the customer element node you just created, create the order element with the orderid attribute value taken from the id attribute in the source document, and with the qty, date, and the amount elements taking their values from the qty attribute and the date and amount elements in the source document.

9. Save your changes to the file.

10. Using an XML editor that supports XSLT, generate a result document named **customers.xml** in the **xml05 ▶ case3** folder.

11. Use your XML editor or another XML processor to verify that the **customers.xml** is well-formed and valid based on the DTD in the **customers.dtd** file.

**CREATE**

## Case Problem 4

**Data Files needed for this Case Problem:** **campingtxt.xml**, **campingtxt.xsl**

**Cairn Camping Store** John Blish manages accounts for the **Cairn Camping Store**, a hiking and camping store based in Redmond, Washington. Information from the sales and accounts database is often written to XML files for use with other programs and applications. John often works with an XML file containing a listing of customers and their recent orders. He would like to transform the contents of this document into a web page that can be easily viewed in his browser. One possible design for his report is shown in Figure 5-42.

**Figure 5-42** Customer orders web page



Name	Evans, Terry
Address	641 Greenway Blvd. Mount Hope, OH 44660
Customer ID	c5391

8/6/2017	54814	
Item No.	Description	Qty
sb8502	Sunblock SPF 30 (Hiking Size)	6
br9002	Bug Repellent (Deep Woods)	2
sb6601	Solar Battery Recharging Unit	1

8/5/2017	53003	
Item No.	Description	Qty
hp7814	Fiberglass Light Hiking Poles (Spring Adj.)	1

8/1/2017	52517	
Item No.	Description	Qty
tr8140	Trail Mix (Pouch)	5
bb7117	Blister Patches	3
wb7133	Insulated Water Bottle	2
bl2815	Boot Laces (Medium)	1
fa8442	First Aid Kit (Pack Size)	1
gps1015	Zendo GPS meter	1

Design an XSLT style sheet to write the HTML code for John's report. Complete the following:

1. Using your text editor, open the **campingtxt.xml** and **campingtxt.xsl** files from the **xml05 ▶ case4** folder. Enter **your name** and the **date** in the comment section of each file, and save them as **camping.xml** and **camping.xsl**, respectively.
2. Go to the **camping.xml** file in your text editor. Take some time to study the content and structure of the document. Note that every customer can have multiple orders on file and within each order are multiple items. Attach the **camping.xsl** style sheet to this XML document. Close the file, saving your changes.
3. Go to the **camping.xsl** file in your text editor and begin designing your XSLT style sheet. John wants the report to include the following features:
  - The name of the store as a main heading.
  - A customer ID table providing each customer's name, address, and ID, with customers listed alphabetically by customer name.
  - Order tables following each customer ID table with the order information for that customer; the order tables are listed in descending order by the order ID.
  - Each order table should include the date of the order and the order ID.
  - Each order table should list the items purchased with the items purchased in the largest quantities listed first. If two products have the same quantity of items ordered, the products should be arranged alphabetically by the item ID.
4. The web page code can be written to HTML5, XHTML, or HTML4 standards. Include the appropriate **output** element specifying how to render the result code.
5. The layout of the page is up to you. You may attach an external style sheet of your own design to make your report easier to read and interpret.
6. Generate your result document using either an XML editor or your web browser. Verify that the page layout and content fulfill all of John's criteria.