XML Programming: Web Applications and Web Services with JSP and ASP

ALEXANDER NAKHIMOVSKY
TOM MYERS

Apress™

XML Programming: Web Applications and Web Services with JSP and ASP
Copyright ©2002 by Alexander Nakhimovsky and Tom Myers

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-003-1

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Slava Paperno
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Erin Mulligan
Copy Editor: Tom Gillen, Gillen Editorial, Inc.
Production Editor: Kari Brooks
Compositor: Impressions Book and Journal Services, Inc.
Artist: Kurt Krames
Indexer: Carol Burbo
Cover Designer: Kurt Krames
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the authors nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

To the memory of my parents. —ADN

To my great^8 *grandfather, Daniel Shipp (1634–1711). —TJM*

Contents

About the Authors ix
About the Technical Reviewer x
Acknowledgments xi
Introduction xiii

Chapter 1 Welcome to XML 1

What Is XML? 2
XML Examples 15
XSLT Programs and XPath Expressions 20
Stylesheet Languages and Browsers 30
Conclusion 39

Chapter 2 Well-Formed Documents and Namespaces ..41

HTML, XML, and XHTML 41
XML Documents Without a DTD 48
Names and Namespaces 52
XLink Attributes and XLink Graphs 60
An XLink Example 65
Namespace Controversies and RDDL 74
Conclusion 79

Chapter 3 DTDs and Validation 81

DTDs and Validation 81
DTD Syntax and Examples 88
General and Parameter Entities 93
DTD Modification and Reuse 102
XHTML Modularization and XHTML Basic 107
What's Wrong with the DTD? 120
The DTD, the XML Document, and the Infoset 120
Conclusion 123

Chapter 4 XML Parsing 125

Basic SAX Programming 126
SAX Filters 143
SAX Parsing for Non-XML Data 157
DOM Programming 169
Traversal Interfaces 182
Book Picker as DOM Builder 186
Conclusion 193

Chapter 5 XPath, XSLT, and XLink Processing 195

An XLink Application: Creating and Using a Linkbase 196
The XPath Language and Data Model 198
XPath Data Model and the Node-Set Data Type 208
Path Expressions 214
XSLT Processing Model 223
Parameters, Variables, and Result Tree Fragments 234
Named Templates and Recursion 236
The Code of the XLink Application 242
Conclusion 253

Chapter 6 More XSLT: Algorithms and Efficiency 255

Specific Patterns and Timing 256
Distinct Nodes and Keys 259
Grouping and Tables 263
Converting Flat to Hierarchical Structure 273
List Processing and Recursion Depth 280
Generating Large Data Sets 284
Extension Functions 293
The Longest Verse, Revisited 297
Conclusion 307

Chapter 7 XML Repository 309

The Structure of XML Data 310
The Structure of the Database 319
The Structure of the Application 320
Driver, Database, Connection, and Statement 331
Query Implementations 1: UPDATE Queries 339
Query Implementations 2: Refset Actions 353
Conclusion 358

Chapter 8 RELAX NG and XML Schema 361

RELAX NG History and Current Condition 362
RELAX NG Overview 369
Modularity and Reuse 383
The datatypeLibrary and XML Schema Data Types 392
XML Schema Part 1: Structures 401
Conclusion 424

Chapter 9 Web Services 425

What's a Web Service? 425
An Example 431
Client Variations 448
Overview of SOAP 1.2 456
XML Encoding and RPC Conventions 462
The Office Equipment Web Service 467
Publish-Find-Bind with UDDI 479
Conclusion 489

Appendix A Installation Guide 491

Version Updates 492
The Java Framework 492
The Windows Framework 495
If Space Is at a Premium 498
Database Connectivity 498
Large Data Files 499
Web Services Examples (Chapter 9) 499
Additional Platforms 500

Appendix B Web Applications 501

General Framework 501
CGI 502
Improvements to Backend Processing 503
ASPs 503
Java Servlets and JSPs 505

Appendix C HTTP Protocol 509

URIs, URLs, and URNs 509
Overall Operation 510
Request Commands (Methods) 513
Server Response Codes 514

Appendix D Online Resources 517

Standards 517
Sources of Information 519
Sources of Software 522
Keep Looking 523

Appendix E Troubleshooting in JSP 525

Looking at Servlet Code for JSP 525
Error Pages 527
Writing Modular JSPs 529
Classpath Problems in Java/JSP 530

Index 531

About the Authors

Alexander Nakhimovsky

Alexander Nakhimovsky received an M.A. in mathematics from Leningrad University in 1972 and a Ph.D. in linguistics from Cornell in 1979, with a graduate minor in computer science. He has been teaching computer science at Colgate University since 1985. He is the author (jointly with Tom Myers) of several books and book chapters, including *JavaScript Objects* (1999), *Professional Java XML Programming* (1999), and three chapters in *Professional JavaServer Programming, J2EE edition* (2000) (all three from WROX), as well as books and articles on linguistics and AI.

Tom Myers

Tom Myers studied physics in Bogota and Buenos Aires before receiving his B.A. from St. John's College, Santa Fe (1975) and a Ph.D. in computer science from the University of Pennsylvania (1980). A software developer and consultant, he has been working mostly on Java/XML projects for the past few years; some earlier research in parallelism and in functional programming languages seems to be coming back to life, within XSLT. In addition to joint publications with Alexander Nakhimovsky, he is the author of a book and several articles on theoretical computer science.

About the Technical Reviewer

Slava Paperno

Slava Paperno has a degree in English from a major university in Russia, and a degree in Russian from an ivy league school in the United States. He has published translations, textbooks, and dictionaries, and has produced television documentaries. He teaches IT seminars for librarians and is the principal author and programmer at his multimedia company, `http://www.lexiconbridge.com`.

Acknowledgments

We would like to thank the team of Apress professionals who guided this book from conception to completion: Gary Cornell, Jason Gilmore, Erin Mulligan, Grace Wong, and Tom Gillen—the meticulous stylist. Special thanks go to our technical reviewer, Slava Paperno, who was relentless in his determination to make us do our best. Emory Creel helped with testing the code on the Linux platform. We are solely responsible for the remaining errors in judgment and execution.

Introduction

What Is This Book About?

This is a book about XML technologies. It does not assume any previous knowledge of XML, and it does not spend much time on how XML can be displayed in the browser or in other media. Our interests are in how XML is used within distributed applications. The big themes of the book are as follows:

- data representation with XML and XLink
- programming in XSLT
- using DOM and SAX APIs
- using XML data with relational databases
- building distributed applications with SOAP, WSDL, and UDDI

One thing to notice is that none of these depend on any specific programming language as substructure: DOM and SAX are available in many languages, XSLT processors are written in (at least) Java and C++, and SOAP is for sending XML messages between endpoints that can be programmed in anything whatsoever.

Who Is This Book For?

This book is for programmers—people who are used to reading, writing, and debugging code. We assume that you have programmed in Java, Visual Basic, C, or C++. We assume some familiarity with HTML/CSS; if you don't have experience in any of these areas, look over a quick tutorial. We assume that you have seen, if not written, JavaScript (ECMAScript, JScript) code. Finally, we assume that you are used to dealing with more than one language: if you mostly write in VB, you can read a commented piece of Java or C++ code and understand what is going on. Reading is easier than writing: even if you have not done much of it, reading code in a less-familiar language is a learnable skill, and it's a skill worth developing because techniques and algorithms are frequently transferable from one language to another. We have written Java code based on VB examples as well

as the reverse, and we spend a lot of time reading through open-source XML systems. You'll be downloading quite a few of these to work with our examples, and you can simply take the binaries but the source is right there for you to study. (It helps in debugging, too.)

To avoid dependence on any one programming language, we have provided several possible frameworks for working with XML. The same XML-processing task can frequently be performed using any of the following:

- JavaScript code within an HTML page.
- a command-line program.
- a Web application using Apache Tomcat and JavaServer Pages (JSPs).
- a Web application using IIS/PWS and Active Server Pages (ASPs), written either in VBScript or JScript. (In one case, we use VB compiled into an ActiveX object invoked from the browser.)

Most of our code consists of Web applications. It is distributed as two zip archives that unzip into the wwwroot directory of the IIS web server and the webapps directory of the Tomcat Web server. (Appendix A contains detailed installation instructions.) Once you have set up the Web servers, you can click your way through the book's examples, seeing code listings, XML data, and active examples; you can then see how ideas work in different contexts. You don't have to understand all the details of the programs that run the examples, but we are confident that you will understand the essentials of all of those programs, and at least several will be completely understandable so you will be able to experiment with them.

What Do You Need to Use This Book?

In quick summary, here is what you need:

- The Java platform, that is, Java Development Kit (JDK) version 1.3 or later. (The book's code was developed with 1.3.1; the current version as we write this introduction is 1.4.)
- The Tomcat Web server that runs servlets and JSPs.
- On Windows, an IIS Web server that comes with ASP support.

- An XML parser and an XSLT processor.
- For SOAP, a SOAP framework; for UDDI, a Web services development toolkit.

Appendix A provides complete and detailed installation instructions for all the programs used in the book. We suggest that you do at least the basic installation (for the first six chapters) before you start reading.

The Book's Contents in Brief

The book consists of nine chapters. The first three chapters introduce the basic notions and main XML technologies: XML as a framework for defining markup languages, well-formed documents, DTDs and validation, and the infoset as the XML data model. The first chapter includes a small sample of XPath and XSLT, the second chapter introduces namespaces and develops examples with XLink and XPointer, and the third chapter goes into more-advanced uses of DTDs and the XHTML modularization framework.

Chapters 4 and 5 go into programming with XML data. Chapter 4 shows DOM and SAX, the two APIs for working with XML in general-purpose programming languages. Within the SAX section, we show how to build pipelines of SAX filters and how to use SAX for converting non-XML data to XML. Within the DOM section, we show both the basic DOM interfaces and the more recent Traversal interfaces. Finally, we show DOM and SAX working together, with SAX pruning the data before a DOM tree can be built in memory.

Chapter 5 gives an in-depth introduction to XPath and XSLT, including the XPath data model, the XSLT processing model, push-and-pull programming patterns, and the use of recursion. In the end, it builds a substantial XSLT application that processes XLink graph structures in a general way.

Chapter 1 through 5 should probably be read in order. The rest of the book goes into three different directions that can be pursued independently. They are more-advanced uses of XSLT (Chapter 6), XML and relational databases (Chapter 7), and Web Services (Chapter 9). Chapter 8 introduces two grammar formalisms to replace DTDs: RELAX NG and XML Schema. The XML Schema material is needed for the Web services material of Chapter 9; otherwise, Chapters 6 through 9 can be read in any order.

Chapter 6 shows more-advanced uses of XSLT (use of keys, grouping and forming tables, and generating large data sets). Throughout the chapter, we show techniques for writing efficient XSLT code, and provide tools for measuring code efficiency.

Chapter 7 shows how XML data can be stored in a relational database and retrieved using a combination of SQL and XPath queries. It develops a substantial

application, the largest in the book, whose XML data consists of base-level records and annotations on those records and other annotations. The base-level records are descriptions of resources in RDF and Dublin Core; annotations are constructed with XLink.

Chapter 8 presents two recent formalisms for specifying XML languages: RELAX NG and XML Schema. We contrast the layered approach of RELAX NG, which cleanly separates validation from augmenting the infoset of a document with additional information from the grammar, with the more monolithic approach of XML Schema. An important part of the chapter is a section on XML Schema Part 2 that defines a system of simple (without internal parts) data types. That system of data types is completely separable from XML Schema Part 1 (structured types) and used in other contexts. For instance, it is used in Web services, and specifically SOAP, for defining the data types of arguments and returned values in a remote procedure call (RPC).

Finally, Chapter 9 introduces and builds examples of Web Services. We explain what they are and what they are not, and we present SOAP in considerable detail. We walk through the steps of converting a Java class into a SOAP server, and we show how SOAP clients for that server can be generated from a WSDL description of the service. We show three different clients for the same server: a Java client, a Microsoft-specific JScript client, and a standard ECMAScript client that invokes the service via a JSP. A more complex service shows the SOAP XML encoding of structured data (using SOAP-specific `struct` and `array` constructs). In conclusion, we introduce UDDI and show how to register a service with the local UDDI registry, and retrieve and modify its properties.

On the Horizon

If we were to write this book again a year from now, we would make the following changes:

- We would shorten the first three chapters, because the knowledge of basic XML will be more widespread.

- We would keep the XSLT material pretty much as is, except updating for later versions of the specifications. (*XPath 2* and *XSLT 2* are in the works.)

- We would keep the SAX and DOM material pretty much as is, except updating for later versions of the specifications. We would also add material on "pull" processing in .NET and in Java, and more generally on .NET XML processing.

- We would significantly expand database material, adding native XML databases and, possibly, XQuery (the XML Query Language) currently under development at W3C.

- We would also significantly expand Web services material, reflecting all the changes between now and then. As we explain in Chapter 9, this area is likely to undergo big changes before it settles into a stable set of standards.

- We would add material on the Semantic Web and uses of AI (artificial intelligence) in organizing the Web. In particular, we would spend more time on RDF (currently undergoing significant revisions), and add material on ontologies and Topic Maps.

In terms of software frameworks, this book has much more Java/JSP code than it does VB, VBScript, and ASP code, reflecting two factors: Java dominance in XML processing and the added value of the open-source frameworks, where we find our own productivity often improved by access to the system's inner workings. The added value of open source will remain a factor, but, otherwise, a year from now Java and .NET would receive more equal time and space.

About Java Code in the Book

Java code of the book mostly consists of JSPs. Purists may criticize us for putting too much code into JSPs instead of Java classes that much shorter JSPs instantiate. Our response is three-fold. First, none of our JSPs is longer than 200 lines of code. Second, all the longer ones are found in a prototype system (the rdb example of Chapter 7) that uses an RDBMS to store XML data. Although we agree with the principle of information hiding in a mature system, a prototype system such as rdb may actually benefit from a more open framework in which all decisions for future modification can be made in one place. Third, we do divide up the code into classes and methods, using method and class definitions within JSPs extensively. (Note for the Java programmer: classes defined in a JSP end up as local classes of the resulting servlet, but this is immaterial to how the class is written or used within the JSP.)

In addition to easy visibility, the advantage of using JSPs is that they serve as a rapid application development framework: no manual recompilation by the programmer is needed. The development cycle is (1) change code, (2) save, and (3) reload the page in the browser. We have found this to be an excellent environment for developing and experimenting with the book examples.

Using JSPs as a development environment does present specific issues in debugging. We discuss them in Appendix E, "Java/JSP Troubleshooting."

The Book's Code and Customer Support

The book's code is downloadable from `http://www.apress.com`. As we said, it consists of two zip archives that unzip into the wwwroot directory of the IIS Web server and the webapps directory of the Tomcat Web server. Appendix A has detailed installation instructions.

Most of our code consists of Web applications, all of them reachable from index.html. Some of the links show code listings, and others actually run the examples. If you encounter any problems either installing or running the examples of the book, you can send email to the mail list for the readers of this book. Instructions on joining that list will be included in the readme.htm file that's distributed with the book's code.

Formatting Conventions Used in the Book

To ease the reader's burden, this book employs several typographical conventions:

- Filenames and directory paths appear in the regular chapter font.
- All items of code appear in a monospaced font, as do all other code-related items, such as attributes, key combinations, elements, functions, parameters, tag names, and the like.
- URIs, URLs, and other Internet addresses appear in this same code font.
- Keywords, queries, and the names of content models appear in all uppercase.

CHAPTER 1

Welcome to XML

XML IS THE CENTRAL TECHNOLOGY of the Internet. When it first emerged in 1997 to immediate and massive media attention, reasonable people could disagree as to whether it was for real or just another overhyped acronym. Those disagreements are over: XML is for real. Initially intended as a technology for structured and linked documents, it has embedded itself deeply into the fabric of distributed Internet applications: XML is used to describe data formats, data types, data transformations, data linking, data transfer, and data processing.

Although the world of XML is large and growing, its foundations remain unchanged. These foundations are set down in two documents—*XML 1.0* and *XML Namespaces*—both of which are recommendations issued by W3C, the World Wide Web Consortium. W3C, in case you have not visited their Web site yet, is the organization that is primarily responsible for the infrastructure of the Web, including XML. Its recommendations are de facto standards. All of the W3C recommendations, together with working drafts and other kinds of technical reports, can be found at `www.w3c.org/TR`.

In the first three chapters of this book, we'll cover the foundations of XML and much more. Although introductory, they will feature examples that you can test and experiment with. These examples will go beyond the foundations, but we'll try to make them readable and self-explanatory. All examples are presented within a framework that makes it possible to experiment with them even before the framework is completely explained.

In outline, this chapter proceeds as follows:

- XML and W3C
- XML document as linear text and tree-structured data
- navigating XML trees with XPath and transforming them with XSLT
- displaying XML in the browser

What Is XML?

It is customary to start a book on XML with a brief definition that packs a lot of wisdom into a few well-chosen words. Here is our version. We will spend the rest of this section explaining what it means.

XML (which stands for *eXtensible Markup Language*) is not really a language but a framework for defining and using markup languages. Markup languages are used for creating units of information called *XML documents*, which have two standard representations: as a linear text with markup and as a tree data structure.

The interpretation of XML languages is unconstrained by XML itself: they can be used for describing data, programs that process data, communication protocols for transmitting data, or anything else under the sun. The interpretation of an XML language is entirely up to the application that uses it.

Now we have to explain why XML is not a language and what a tree looks like.

XML Languages and Documents

Any language has two components:

- v*ocabulary:* What are the words of the language?

- *syntax:* How do those words hang together?

If the definition of the language does not say anything about interpretation ("What does it mean?"), then we have a *formal* or *uninterpreted* language. Human languages and programming languages are interpreted, whereas XML languages are formal languages.

Different XML languages have different vocabularies and different syntax, but they all share some general constraints. These constraints mostly serve to ensure that an XML document, in any XML language, describes a piece of tree-structured data. They also ensure that it is easy for an XML processor (also known as an *XML parser*) to check the document for correctness and construct the corresponding tree structure in a standard form.

What's a Tree?

A tree is a connected set of nodes and a parent-child relationship defined on them. One special node is called the *root*. Every node except the root has exactly one parent node, and the root is the only node that doesn't have a parent. In computer science (as in genealogy), trees are always drawn upside down, with

the root on top and leaves at the bottom. If you start from any node that is not a root, go up to its parent, and continue up the tree, sooner or later you get to the root. The nodes you encounter along the way are called the *ancestors* of the node you started from. The root is an ancestor of all nodes in the tree, and all nodes in the tree are its *descendants*. If node P is the parent of node C, then C is usually called a *child* of P. Children of the same node are called *siblings*. Nodes that have no children are called *leaves*.

Grammars, Parsers, and Syntax Diagrams

To define a language is to define its vocabulary and syntax. Defining a vocabulary is easy: just list all the words. Syntax is defined by writing down rules, and a collection of syntactical rules is called a *grammar*. Several standard notations are commonly used for writing grammar rules, and XML uses a notation called *DTD* (*Document Type Definition*). We will see examples shortly.

A program that takes a text and checks the correctness of its syntax is called a *parser*. A parser does not simply return a Boolean answer ("correct" or "not correct"): if the text is grammatically correct, it builds an internal representation of its syntactical structure. Such internal representations are called *syntax diagrams*. For XML documents (as well as computer programs), syntax diagrams form a tree structure.

An English-Language Example

Here is a simple example from English, to illustrate some of these concepts. The first sentence below is grammatically correct and easy to interpret; the second uses English syntax but is nonsensical; the third has its syntax scrambled; and the fourth uses nonexistent words but is still recognizably a sentence because it begins with a capital letter and ends with a period.

- Healthy well-fed children sleep peacefully.
- Colorless green ideas sleep furiously.
- Well-fed sleep children peacefully healthy.
- Kannerva poniiatikka oferaduli sarenaa.

The last item on this list points to a feature of (written) languages that is easy to miss: they have delimiters (punctuation marks and spaces) that separate words, sentences, and paragraphs. Given a text in any European language, you

can easily identify the beginning and end of every sentence and paragraph, because the delimiters are the same. If somebody asked you to "parse" such a text and build a tree structure of paragraphs and sentences (paragraphs are children of the root, and sentences are children of paragraphs), you could do it without ever consulting the grammar or the dictionary. This assumes certain rules: paragraphs don't overlap, and no sentence starts in one paragraph and ends in another.

The *XML 1.0* document is very much about delimiters, defined so that you can parse an XML document into a tree structure even if you don't know the vocabulary and the concrete grammar of that particular XML language. However, unlike the punctuation conventions of western languages, XML gives you unlimited flexibility in defining the shape and depth of your tree, and the labels you can put on its nodes.

XML Documents and Markup Languages

An XML document consists of text data and markup. The markup indicates the syntactical structure of the document. Listing 1-1 shows a simple example.

Listing 1-1. Hello, XML

```
<!-- Our first example (and this is a comment) -->
<encounter>
  <greeting>Hello, XML!</greeting>
  <response>Hello, what can I do for you?</response>
</encounter>
```

This XML document contains three elements: `encounter`, `greeting`, and `response`. Here is the greeting element:

```
  <greeting>Hello, XML!</greeting>
```

An element consists of a start tag, the element's content (which can be empty), and an end tag. A start tag minimally consists of the "<" character, a tag name, and the ">" character. The tag name can be followed by attribute declarations. An end tag consists of the character sequence "</", followed immediately by the tag name and the closing bracket. (See Figure 1-1.)

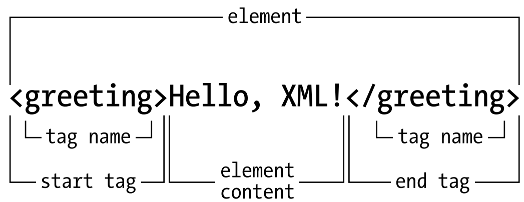

Figure 1-1. An element and its tags

What is the tree structure of the document shown in Figure 1-1? In XML, the parent-child tree relationship corresponds to how elements are nested within each other in the linear text. In our example, `encounter` properly contains `greeting` and `response`; therefore, in the tree, `encounter` is the parent of `greeting` and `response`, whereas `greeting` and `response` are children of `encounter` and siblings to each other. Leaf elements are either empty or contain only text.

Figure 1-2 is a syntactical diagram for Listing 1-1. It follows two conventions:

- There is a root node that is parent to the root of the element tree and to top-level comments, if there are any.
- The text content of an element is wrapped in a text node.

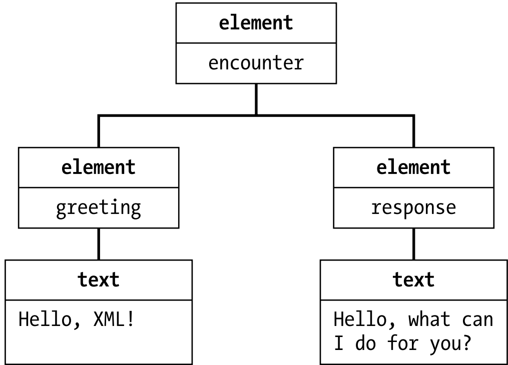

Figure 1-2. Tree diagram for Listing 1-1

The conventions of these diagrams come from the XPath specification, which is one of several standards that specify how XML trees are represented and used in computer programs.

Tree Representations of XML

XPath is a language for navigating a tree and selecting a set of nodes for processing. Using XPath, it is easy to say such things as "Give me the text content of the second child of my next sibling in the tree" or "Give me all the element nodes in the tree whose tag name is *section* and whose parent's tag name is *chapter*." XSLT (eXtensible Stylesheet Language for Transformations) uses XPath expressions to access sets of nodes in an XML tree and use them for building an output structure. Neither XPath nor XSLT modify their XML data source in any way.

DOM (Document Object Model) is a set of APIs that provide both for access and modification of the input tree from a programming language. DOM has been implemented in many programming languages including Java, ECMAScript, C, C++, Python, and Visual Basic. DOM APIs include such functions as `getParent()` or `getChildren()`. There is considerable overlap between what you can do in DOM and in XPath+XSLT, but XPath is a higher-level language: it is easier to select a set of nodes in XPath than in DOM.

Both XPath and DOM represent tree structures. Because they were developed independently and in parallel, their current versions (XPath 1.0 and DOM 2.0) disagree in minor ways in their tree models of XML data. To eliminate such discrepancies in the future, a new standard called *infoset* specifies what information items from XML text must be preserved in parsing and how they relate to each other. In other words, the DOM tree, the XPath tree, and the text+markup document will represent the same infoset. All future XML standards will conform to the infoset specification, including DOM 3 and XPath 2.0. We will explain shortly how and by whom these standards are developed.

> **NOTE** *When DOM and XPath are synchronized, a standard will be developed for using XPath expressions within DOM so that DOM functions can operate on sets of nodes selected by XPath expressions. Some DOM implementations currently provide this functionality in vendor-specific ways.*

Grammar Rules of XML Languages

Grammar rules of XML languages specify for each element whether it contains text or children elements, or both, and what attributes, if any, it may or must have. For instance, to specify that `encounter` must have two children, `greeting` and `response`, in that order, we would say:

```
<!ELEMENT encounter (greeting, response)>
```

To specify that the `greeting` element contains only text, we would use the keyword PCDATA (Parsed Character DATA):

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

A complete set of such rules forms a grammar (a DTD) for an XML language.

Types of Element Content (Content Models)

The element content can be of four kinds called *content models*: text-only, elements-only, a mix of text and elements, and empty. The `greeting` and `response` elements have text-only content, and the `encounter` element has element-only content.

Text and elements can be mixed together as in

```
<p>This is an example of text broken into long <keyword> paragraphs</keyword>,
with some <keyword>words</keyword> and <keyword>phrases</keyword>
marked up for <em>special treatment</em> (perhaps displayed in bold face,
<span thickness="bold">like this<span>), and some
<keyword>items</keyword> provided with comments.</p>
```

An element's content can be empty as in

```
<br></br>
```

Even empty XML elements must have an end tag or else be written in a special form that clearly indicates the beginning and end of the element: `<br/>`. This may seem like a trivial detail, but in fact it has very important consequences: it makes the parser's task much easier.

Parsing XML

A parser for a programming language knows only that one programming language: a Java parser cannot do C++. Moreover, to parse a program, the parser has to consult the grammar of the language. But XML parsers are different on both counts: the same parser can do all XML languages, and, to parse a document, the parser does not need to know the grammar of its markup language. A general XML requirement is that the elements of an XML document must form a tree and that the tree structure of elements must be clearly shown in markup.

In practice, it means that tree conditions must be met:

- There must be an element that contains all other elements. This is the root of the element tree (`encounter` in our example).
- Start and end tags must be properly nested; overlapping elements are not allowed.
- All elements, including empty elements, must have both the start tag and the end tag.

Thanks to these requirements, all the parser needs to construct the syntactical tree is a stack to push a start tag on and pop it off when the matching end tag is found. If in the end the stack is empty, the document is well formed. More precisely, the procedure would be as follows:

1. Start by creating the root node and make it the current node.
2. When a start tag is encountered, create a child of the current node and make it the current node. Put the start tag name on stack.
3. When an end tag is encountered, check to see that its tag name is the same as the name on top of the stack. If it is not the same, declare failure and exit. If it is the same, close the current node (that is, pop the tag name off the stack) and make its parent the current node.
4. If in the end the stack is empty, declare success, and return the tree. If it is not empty, declare failure and exit.

XML parsers within XML-aware browsers follow this procedure. A useful exercise is to create a document with a missing end tag and open it in such a browser. Try to predict, using the procedure, what error it will report.

Well-Formed Documents and Parser Attitudes

The three requirements of the preceding section are some of the "well-formedness" conditions that are listed in *XML 1.0* and *XML Namespaces*. An XML document must satisfy all of these conditions. Documents that satisfy all well-formedness conditions are said to be *well formed*. Documents that are not well formed must be flatly rejected by the parser. A standard battery of tests has been designed to make sure that parsers catch all well-formedness errors; a parser that does poorly on the tests receives a bad grade and bad press.

> **NOTE** *The test suite is developed and maintained by OASIS (Organization for the Advancement of Structured Information Standards,* `www.oasis-open.org`*). Other valuable resources are available at that site.*

Why Is XML Great?

At this point, we are in a position to offer a couple of reasons why XML is great. One reason is that XML makes it easy to agree on a common language or data format. A common language is the main prerequisite for cooperation.

Another reason why XML is great is because it is very easy to switch between the linear text and the syntax tree view of an XML document. XML parsers are standard, high quality, ubiquitous, and free, and they can perform the switch both ways without any loss of significant information. (What's significant? We will explain in Chapter 2.)

The reason we may want to switch from text to tree and back is because a tree data structure is easy to work with but linear text is easy to send over the network. With XML, it is easy to construct networks of cooperating computer programs that receive XML text over the network, parse it into its internal representation, perform some computations on it, convert it back into linear text and send it over the network to another program for further processing.

Adding the two together, we can say that XML is a major enabling technology for cooperation, both between human agents and between computer programs (interoperability).

More on Parsing: XML vs. Programming Languages

Converting linear text to a binary is, of course, nothing new: this is what happens every time we parse a program. It is useful to compare and contrast parsing a program and parsing XML.

- Parsers for programming languages are difficult to write.
- The resulting binary objects are parser and platform specific.
- Transition from binary to textual form (disassembly) is hard and frequently illegal; certainly, no standard APIs are available for doing that.

By contrast, simple XML parsers that only check for well-formedness and construct a syntax tree are easy to write, and their output is standard, platform independent, and easy to convert back to linear text form.

XML Is Just Syntax, No Interpretation

We have already mentioned another important point of contrast between a programming language and an XML language. Expressions in a programming language have predefined meaning (interpretation). A programming language parser is usually part of a larger program (compiler) that applies rules of interpretation to the syntactical tree produced by the parser. On the other hand, the role of an XML parser is more modest: it produces a tree and submits it to an application. In other words, XML itself doesn't mean anything until it is processed by some application that "interprets" XML data. This is XML's great strength: different applications can assign different meanings to the same XML data. (As one example of this general principle, different rendering engines can render the same XML data in completely different ways.)

To illustrate this point, compare the following two expressions. The first is a LISP expression, which the language processor parses to build a syntax tree. But it doesn't stop there: it proceeds to evaluate the expression according to the predefined semantics of operators and arguments. The meaning of this LISP expression is 12.

```
(times
    (+ 3 1) ; evaluates to 4
    (- 5 2) ; evaluates to 3
) ; evaluates to 12
```

Now consider an XML document of a very similar structure:

```
<times>
  <plus><num>3</num> <num>1</num> </plus>
  <minus><num>5</num> <num>2</num> </minus>
</times>
```

Any XML parser will parse it, build its syntax tree, and pass it on to whatever application invoked the parser. This document has no predefined meaning outside the application. XML is just syntax.

XML Data, the Parser, and the Application

The XML parser mediates between XML data and the application that interprets and uses the data. Their mutual relationship can be described in a diagram, as shown in Figure 1-3.

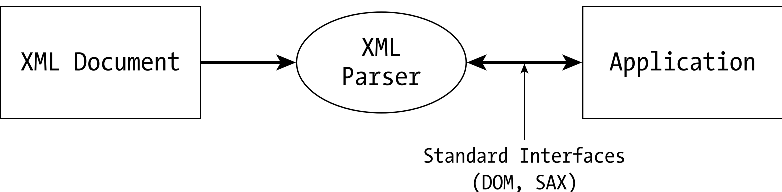

Figure 1-3. XML data, the parser, and the application

The behavior of an XML parser is heavily regulated on both sides of Figure 1-3. Its treatment of the XML document is precisely specified in *XML 1.0*, in the section on Conformant Processors (enforced by OASIS conformance tests). Its obligations towards the application are precisely specified in two standard APIs: DOM and SAX (Simple API for XML). SAX is yet another way to represent an XML tree—as a sequence of events and callbacks. We will expand on this in great detail in Chapter 4.

XML As a Serialization Format

Because it is easy to go back and forth between the linear text and the tree-structured view of XML, it is difficult to say which one is more important or primary. Some people think of XML documents as marked-up texts and syntactical trees as supporting machinery for working with texts; others think of XML documents as tree structures of a special kind and linear text as a serialization format for XML data. The first view is more common among those who work with XML documents created by humans for humans, and use markup for publishing or display. The second view is more common among those who work with XML generated by programs for programs; such XML can go through a long and productive life cycle never encountering an angle bracket. A true XML professional can switch between those two views with the same ease as an XML parser switching from text to tree and back.

To make such distributed processing possible, we need standards, both for text representation and for (working with) tree structure. The next section presents some of those standards and explains who writes them.

XML Standards and W3C

XML as text is defined in *XML 1.0* and *XML Namespaces*, the two standards that are the untouchable foundation of XML. Each remains in its initial release version (*XML 1.01* is a minor edit), and no new versions are in the works. By contrast, XML as tree has been defined several times in different standards (we

have already mentioned XPath, DOM, and infoset), each of which is undergoing further development. This is not surprising: XML tree representations are used in programming and evolve in response to programming needs. (See Figure 1-4.)

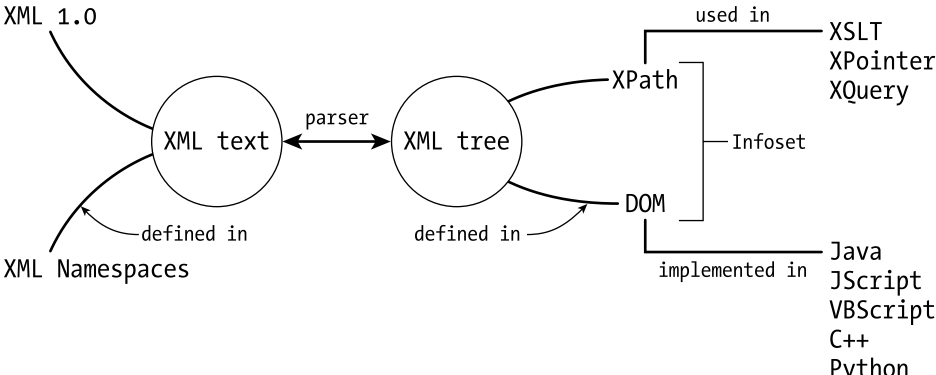

Figure 1-4. Standards for XML text and XML tree

XSLT uses XPath expressions to access sets of nodes in an XML tree and uses them for building an output structure. XPath is also used within the XML Pointer specification (*XPointer*) to specify the source and target of an XML Link (*XLink*).

If XML data is used by an application written in a general-purpose programming language (rather than XSLT), the application uses either DOM or SAX standard interfaces. Of all specifications we have mentioned, SAX is the only one that has not been developed by W3C.

The Essence and Alternative Views of XML

This may be a good point to repeat yet again an important idea that we have already introduced and to which we will return many times throughout the book: The essence of XML is that it is hierarchical (tree-structured) data. Its structure can be represented textually as markup, by tree structures in computer memory, or in some other way. As we proceed, we will learn several ways to represent XML data: XPath, DOM, infoset, and SAX. The main point is that they are all alternative views of the same hierarchical data, and they are all regulated by open standards. As a result, computer programs that work with XML data can use any of these alternative representations as input and output, even if the data comes from or goes to a completely unknown agent on the Internet. This creates rich possibilities for constructing distributed Web applications—provided the standards are indeed developed and enforced.

W3C

Who develops the standards? The answer is (mostly) W3C, the World-Wide Web Consortium. W3C was initially created to standardize HTML, a markup language that predates XML markup languages. HTML was defined by Tim Berners-Lee, using SGML (Standard Generalized Markup Language). It is an international standard approved by ISO (ISO 8879:1986). Like XML, SGML is not itself a language but rather a framework for defining and using markup languages. It was created for the publishing industry without any Internet applications in mind. Another way to define XML is to say that it is a simplified and Internet-ready descendant of SGML.

Initially, HTML was maintained in the same way as other Internet standards: published as an RFC (request for comments) and eventually approved by the IETF (Internet Engineering Task Force). That was okay as long as there was little or no money in it. When HTML became big business, the main browser vendors started pulling it apart, adding proprietary tags and rules, and IETF procedures proved to be too slow to keep up with browser releases. The Web was in danger of fracturing into incompatible browser domains. In response, a group of good people led by Berners-Lee and Michael Dertouzos of MIT created the W3C consortium that has taken upon itself to keep the Web whole. Its first great success was to impose a uniform HTML standard on competing browsers: HTML 3.2 followed by HTML 4.0 and 4.01. Its second great success was to invent XML. One side effect of this invention is that there will be no further releases of HTML: it is superceded by XHTML, a reformulation of HTML in XML. We will compare and contrast HTML and XHTML in the next chapter.

W3C's Procedures and Recommendations

W3C is not a standards body; rather, it is a consortium whose members come from industry, academia, and national governments. The documents issued by W3C are called *recommendations* and, theoretically, can be ignored. However, the high technical quality of its work, its elaborate procedures for consensus building, the combined weight of its members, and the authority of its director make W3C recommendations de facto standards. Because W3C is the custodian of XML and XML has become the definitive technology of the Web, the evolution of the Web is to a great extent determined by the documents moving through the W3C pipeline.

The pipeline is organized as follows. First, a certain area of activity (for instance, XML Query Language) has to be adopted for development by W3C, and a W3C working group is formed. The group's first task is to issue a requirements specification and, where appropriate, a list of use cases. These are followed by a sequence of working drafts (WDs) that culminate in a candidate

recommendation (CR). The release of a CR is a signal to developers that the specification is feature-complete and ready for implementation. The next stage is a proposed recommendation (PR) that takes into account the experience accumulated during implementation and testing. After a final round of comment and revision—and if all goes well—the proposed recommendation is signed by the director and becomes an official recommendation (R). Table 1-1 shows a selected list of XML specifications.

Table 1-1. Some XML Specifications

SPECIFICATION	STATUS	DATE	DESCRIPTION
HTML 4.01	R	December 1999	The last HTML; XHTML is taking over.
XML 1.0	R	February 1998	The foundation on which everything rests. Includes DTD, a language for defining markup languages.
Namespaces in XML	R	January 1999	An important addition to the foundation.
XML Schema	R	May 2001	Language for defining markup languages, to replace or complement DTD.
XPath 1.0	R	November 1999	A language of tree paths to navigate XML data.
XSLT 1.0	R	November 1999	XML Transformation language.
XHTML 1.0	R	January 2000	HTML 4.01 revised to conform to *XML 1.0*.
XHTML Basic	R	December 2000	A subset of XHTML for small devices.
DOM Level 2	R	November 2000	Standard API for working with XML data.
Infoset	CR	May 2001	Information content of XML trees.
XLink	R	December 2000	Language for linking documents, including multidirectional and third-party links.
XPointer	WD	January 2001	Language to specify locations and ranges within an XML document; uses XPath; itself used by XLink.
XQuery	WD	February 2001	Language for querying XML data repositories, patterned after SQL.

In the next section, we will look at a series of examples to show the components of XML documents and introduce the common terminology. The section after that will show a small selection of XPath expressions, to be used in exercises within simple XSLT programs. (XSLT is a language for transforming XML data; we will have more to say about it in the next section, and much more later in the book.)

XML Examples

This section presents a rapid succession of different kinds of XML documents, starting with the very simple and moving on to documents that include DTDs and use namespaces. They will introduce the terminology and serve as a preview for the next two chapters.

Examples Without a DTD

A complete and correct XML document can be as short as

```
<q>What's up?</q>
```

This XML document consists of a single element, but most of them have more structure. Our next example has three elements, one of which has an attribute:

```
<exchange><q tone="informal">What's up?</q><a>The parent node.</a></exchange>
```

XML examples are usually divided into several variously indented lines to indicate their tree structure. We show this one on a single line to emphasize that it is just a linear sequence of characters.

Declaration and Encoding

Many XML documents start with an optional declaration:

```
<?xml version="1.0" encoding="utf-8"?>
<q>What's up?</q>
```

The optional encoding attribute tells the parser how the Unicode characters are encoded as bytes. The problem of byte encoding does not arise with ASCII characters because there are only 128 ASCII characters and each fits within a single byte. The Unicode standard contains 65,536 characters, and at least some of

them have to be encoded as more than one byte. The two most common encodings that all XML parsers are required to support are UTF-8 and UTF-16. (*UTF* stands for *Unicode Transfer Format*.) UTF-16 encodes all characters as a sequence of two bytes, and UTF-8, the default, is a variable-length encoding that uses one, two, or three bytes to encode a character. It is designed so that the ASCII range of characters looks exactly the same in ASCII and in UTF-8; in other words, UTF-8 is backward compatible with ASCII.

Comments and Processing Instructions

XML documents can contain comments and processing instructions (PIs). These can be inserted anywhere in the document except before the declaration or inside tags. (See Listing 1-2.)

Listing 1-2. A Document with a Comment and a PI

```
<?xml version="1.0"?>
<!-- This is a comment. The next line is a PI -->
<?xml-stylesheet href="exchange.css" type="text/css"?>
<exchange tone="informal">
  <q>What's up?</q><a>Nothing much.</a>
</exchange>
```

Theoretically, PIs can be used to invoke an application from within an XML document, but in practice they are used only to refer to a stylesheet (which is explained later in this chapter), as in this example.

XPath Tree Model

We'll use this short document to remind you of the XPath tree view of XML, also known as the *XPath Tree Model*. (See Figure 1-5.) Recall two important conventions:

- There is a root node that is parent to the root of the element tree and to top-level comments and processing instructions, if any.
- The text content of an element is wrapped in a text node.

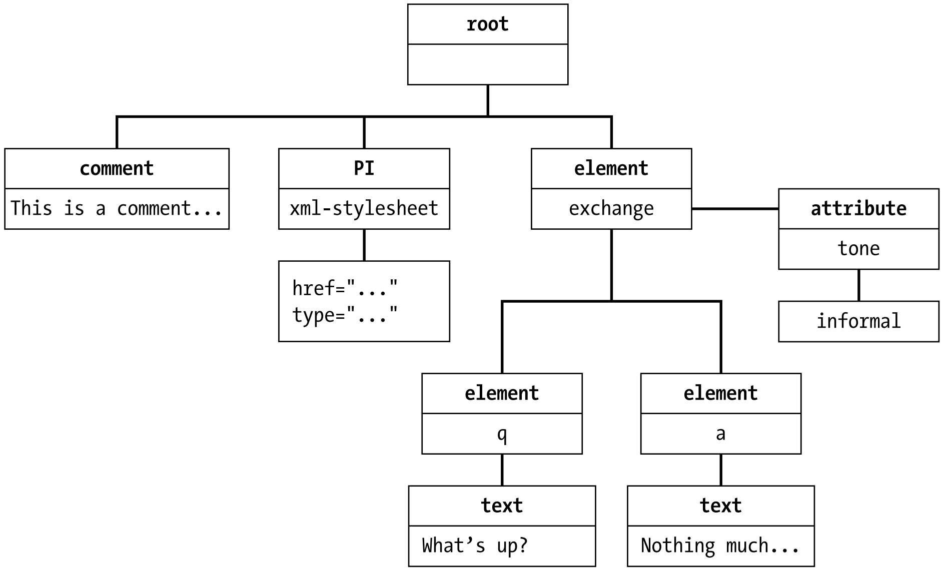

Figure 1-5. An XPath tree for Listing 1-2

Attributes and a DTD

The next example brings in attributes and a DTD. XML attributes work exactly the same way as HTML attributes: they are name-value pairs that are listed in the start tag of an element right after the tag name. Attribute values must be quoted, as in Listing 1-3.

Listing 1-3. A Document with Attributes

```
<?xml version="1.0" encoding="utf-8"?>
<exchange tone="polite">
  <q>What would you like to drink?</q>
  <a>Whatever you have is fine.</a>
</exchange>
```

This is a well-formed XML document, but so is the one shown in Listing 1-4.

Listing 1-4. Same Document, Rearranged

```
<?xml version="1.0" encoding="utf-8"?>
<exchange tone="cloudy with occasional rain showers">
  <a>Whatever you have is fine.</a>
  <q>What would you like to drink?</q>
</exchange>
```

If you want to make sure that questions precede answers, you need a DTD, as shown in Listing 1-5.

Listing 1-5. A Document with a DTD

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE exchange [
<!ELEMENT exchange (q, a)>
<!ELEMENT q (#PCDATA)>
<!ELEMENT a (#PCDATA)>
<!ATTLIST exchange tone (friendly|polite| cold|rude) "friendly">
]>
<exchange tone="polite"><!-- same as Listing 1-3 --></exchange>
```

This declares a document type called *exchange* and defines its elements. The root element is exchange. (The name of the document type and the name of the root element don't have to be the same, but it's common practice to have them so.) The exchange element's content model is children-only, `q` and `a`, in that order. Both `q` and `a` are text-only, which in DTD-speak comes out as PCDATA. The exchange element has a tone attribute whose possible values are listed in the declaration, with "friendly" being the default: if your document does not contain the attribute, the parser will insert `tone="friendly"`. (You can test this in an XML-aware browser. For instance, in Windows, save Listing 1-5 (with the attribute removed) in a file with the .xml extension and open it in Internet Explorer 6. You will see the default inserted.)

External DTDs

You can put the DTD in a separate file and refer to it from the document, as shown in Listing 1-6.

Listing 1-6. A Document with an External DTD

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE exchange SYSTEM "exdtd.dtd">
<exchange><!-- same as Listings 1-3 and 1-5 --></exchange>
```

The contents of exdtd.dtd would be as follows:

```
<!ELEMENT exchange (q, a)>
<!ELEMENT q (#PCDATA)>
<!ELEMENT a (#PCDATA)>
<!ATTLIST exchange tone (friendly|polite| cold|rude) "friendly">
```

Valid Documents and Validating Parsers

If you have a DTD and your parser knows how to use it and you configure your parser so that it does indeed check your documents against the DTD, then the parser will accept Listing 1-3 but reject Listing 1-4. A document that checks out correctly against its DTD is called *valid* with respect to that DTD, and a parser that understands the DTD language is called a *validating parser*.

DTDs are good at design time, and they are useful as a concise description of a shared language. At production time, you frequently do not want to validate because the DTD may not be available or it may reside on a remote server and take an indefinite period of time to download. Even if there is a local DTD, validation is an extra processing step that incurs a performance penalty. For these reasons, validating parsers typically have a settable Boolean flag that turns validation on and off. (We will cover validation in Chapter 3.) As you know, you do not need validation to use XML data because XML processors can parse a document and construct its tree without a grammar.

Document-Oriented vs. Data-Oriented XML

XML documents tend to one of two extremes. At one extreme are documents that are primarily text, with markup inserted occasionally to bring out the text structure and select some text ranges for special treatment. A typical feature of this kind of XML is that its elements frequently have a mixed-content model: text and elements mixed together as siblings (children of the same element).

The other kind looks as if it comes out of a relational database, and frequently it does. It is highly structured, with repeating elements of the same internal content. Text in this kind of XML appears only in the leaf elements of the tree, never at the same level as children elements. See Listing 1-7.

Listing 1-7. Data-Oriented XML

```
<?xml version="1.0"?>
<!-- personal data for people and other kinds of personalities -->
<pdata>
  <person id="CM123" access-level="customer">
    <name>
      <title>Mr.</title>
      <last>Monster</last>
      <first>Cookie</first>
      <middle>C</middle>
    </name>
    <address>
      <street>123 Sesame Street</street>
```

```
      <city>New York</city>
      <state>NY</state>
      <zip>10023</zip> <!-- the zip code is real -->
    </address>
    <bdate>
      <year>1969</year>
      <month>11</month>
      <day>2</day>
      <!-- Official birthday November 2nd; first show was 11/10/69 -->
    </bdate>
    <email>cookie@sesamestreet.com</email>
    <!-- we made that up, but telly@sesamestreet.com is "real" -->
    <favorites>
      <color>red</color>
      <drink>cookie juice</drink>
    </favorites>
  </person>
  <!-- possibly many more persons -->
</pdata>
```

We'll be working with this document in the following XPath and XSLT examples.

XSLT Programs and XPath Expressions

XSLT (eXtensible Stylesheet Language for Transformations) is a programming language for transforming XML data. For historical reasons, it is called a *stylesheet language*, and XSLT programs are frequently called *stylesheets*. In this section, we introduce simple XSLT programs that will help you explore the tree structure of XML data. In addition to presenting new theoretical material, this section is the first hands-on section in the book. It is best read in front of a computer that is set up to explore and experiment with the book's code.

Prerequisites and Setups

As explained in the introduction, all our code runs in the context of a Web application. The setup for working with it consists of two parts: the basic setup for a Web application plus XML tools. The best way for you to get everything ready for hands-on work depends on your background.

- If you have worked with ASP Web applications before, review the section on JSP in Appendix B and proceed to the installation instructions in Appendix A.
- If you have worked with JSP Web applications before, review the section on ASP in Appendix B and proceed to the installation instructions in Appendix A.
- If you have worked with servlet-based Web applications but not with ASP or JSP, review the appropriate sections in Appendix B and proceed to the installation instructions in Appendix A.
- If you have never worked with ASP, JSP, or servlets but are familiar with CGI-based Web applications, review Appendix B in its entirety and proceed to the installation instructions in Appendix A.
- If you have never worked with Web applications of any kind, start with Appendix C, proceed to Appendix B, and finally proceed to the installation instructions in Appendix A.

XSLT Setups

To run XSLT programs, you need an XSLT processor. In Web applications, your choice of processor depends on the server, the operating system, and the programming language used. Here and throughout the book, we primarily work with two combinations of software:

- Apache Tomcat (a combined Web server and JSP processor), Apache Xalan (an XSLT processor), and Apache Xerces (an XML parser)
- Microsoft IIS/PWS (a combined Web server and an ASP processor) and MSXML 3.0 or later (a combined XML parser and XSLT processor)

> **NOTE** *Microsoft and Apache are two major sources for standard XML software (partly because much XML software produced by IBM and Sun ends up at Apache). Their world views and business philosophies are, of course, very different: Apache is open source and primarily Java, whereas Microsoft is closed source and no Java at all. It is all the more remarkable that XML standards clearly show through both frameworks, so that skills learned in one of them are easily transferable to the other—not to mention the even more remarkable fact that, given the same XSLT program and the same XML data, both frameworks produce identical results with a high and growing degree of reliability.*

Running XSLT Programs

An XSLT processor receives an XML data source and an XSLT program on input and produces a result as specified by the XSLT. The result is usually XML, HTML, or plain text. A very common use of XSLT is to convert XML data to HTML or XHTML for display in the browser. All XSLT examples and exercises in this chapter belong to this last category.

In a Web application, references to an XML source and a stylesheet can be entered from an HTML form or typed into the address window. The output is sent back to the browser in the usual way. Assuming that we run IIS/PWS on port 80, the page xx.asp is in wwwroot\xmlp, and the two inputs—hello.xml and hello.xsl—are in wwwroot\xmlp\helloXSL, we invoke the XSLT processor on them using this URL (split into two lines):

```
http://localhost/xmlp/xx.asp?
   xmlUri=helloXSL/hello.xml&xslUri= helloXSL/hello.xsl
```

This, of course, further assumes that xx.asp expects request parameters `xmlUri` and `xslUri`. (See Listing 1-9). Similarly, if we use Tomcat on port 8080, the page xx.jsp is in webapps\xmlp and the two inputs, hello.xml and hello.xsl, are in webapps\xmlp\helloXSL, we invoke the XSLT processor on them using this URL:

```
http://localhost:8080/xmlp/xx.jsp?
    xmlUri= helloXSL /hello.xml&xslUri= helloXSL /hello.xsl
```

with hello.xml being

```
<greeting>Hello, XML world!</greeting>
```

and hello.xsl as in Listing 1-8.

Listing 1-8. The First XSLT Program

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html><head><title>First XSLT Program</title></head>
    <body>
      <h1 align="center"><xsl:value-of select="greeting"/></h1>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The output is shown in Figure 1-6.

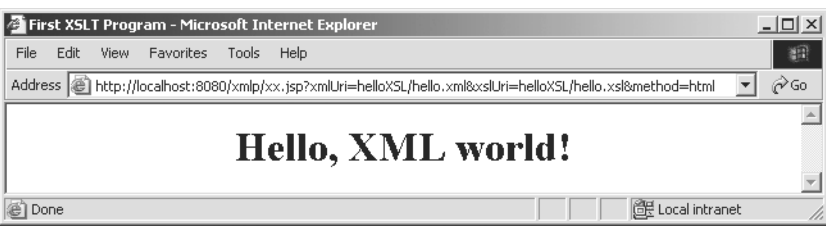

Figure 1-6. Hello, XML world!

As you can see, an XSLT program is an XML document of a special kind. We will explain every detail of our first XSLT program shortly, but first take a look at the code that invokes it.

ASP Code

In outline, xx.asp (Listing 1-9) proceeds as follows:

1. Retrieve the XML file name from Request.
2. Find its absolute path to server root.
3. Create an XML tree (DOM) object called xmlObj.
4. Parse ("load") the XML file to create a tree.
5. Repeat these actions for the XSLT program. (An XSLT program is an XML document.) The program is loaded into an XML tree object called `xslObj`.
6. Transform `xmlObj` using the program stored in `xslObj`; send output to Response.

Listing 1-9. ASP Code to Run XSLT

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
  Dim xmlFile,xmlObj,xslFile,xslObj
  xmlFile = Request.QueryString("xmlUri")
  xmlFile = server.MapPath(xmlFile)
  Set xmlObj = CreateObject("MSXML2.DOMDocument")
  xslFile = Request.QueryString("xslUri")
  xslFile = server.MapPath(xslFile)
  Set xslObj = CreateObject("MSXML2.DOMDocument")
  xmlObj.load xmlFile
  xslObj.load xslFile
  Response.write(xmlObj.transformNode(xslObj.documentElement))
%>
```

JSP Code

The JSP version, xx.jsp, is quite similar, except there are data types, and exceptions get thrown (as shown in Listing 1-10).

Listing 1-10. JSP Code to Run XSLT

```
<%@ page errorPage="error.jsp"
    import="javax.xml.transform.TransformerFactory,
             javax.xml.transform.Transformer,
             javax.xml.transform.stream.StreamSource,
             javax.xml.transform.stream.StreamResult,
             java.io.File"
   %><%
// get directory paths to XML and XSLT files
  String xml=application.getRealPath(request.getParameter("xmlUri"));
  String xsl=application.getRealPath(request.getParameter("xslUri"));
// check that files exist
  if(!new File(xml).exists()) throw new Exception("no file "+xml);
  if(!new File(xsl).exists()) throw new Exception("no file "+xsl);
// obtain a transformer object that implements the XSLT program
  TransformerFactory tFactory = TransformerFactory.newInstance();
  Transformer transformer = tFactory.newTransformer(new StreamSource(xsl));
// apply the transformer to given XML; send output to the out stream
   transformer.transform(new StreamSource(xml), new StreamResult(out));
%>
```

You can use either or both of these pages to work with examples and exercises of this chapter. With a framework in place, we can proceed to the XML-XSLT-XPath substance of the matter.

XPath Expressions

XSLT programs operate by selecting a set of nodes in the input and using the material of selected nodes to construct the output. XPath is a language for selecting sets of nodes.

XPath expressions, by design, look very much like Unix directory paths. We will illustrate them using the XML of Listing 1-7. The expression "`/`" refers to the root of the document tree. The expression `/pdata` refers to the `<pdata>` element. The expression `/pdata/person` refers to all the children elements of the `pdata` element whose tag name is `person`. To refer specifically to the first such element (Cookie Monster), we use `/pdata/person[1]`. To refer to Cookie Monster's last name, we use `/pdata/person[1]/name/last`.

Attributes in XPath

We refer to attributes in a similar way but insert the "@" character before the name of the attribute. To refer to Cookie Monster's id attribute, we say `/pdata/person[1]/@id`. To refer to Cookie Monster himself by his ID, we say `/pdata/person[@id='CM123']`. Read this as "person such that its id attribute equals 'CM123'". In general, the expression in square brackets is a predicate; `/pdata/person[1]` is a convenient shorthand for `/pdata/person[position()=1]`. In this expression, `position()` is a function that returns the position of the node in the currently selected set of nodes. The list selected by `/pdata/person` is, as we just said, all the children elements of the `pdata` element whose tag name is `person`, in their document order.

An XSLT Program

An XSLT program is an XML document. As such, it has a programmatic core and the surrounding XML supports. Take a look at the programmatic core first (Listing 1-11). Our purpose is to produce an HTML page that says "Hello, Mr. Monster", using Listing 1-7 as XML data input.

Listing 1-11. An XSLT Program, helloXSL/helloSelect.xsl

```
<xsl:template match="/" >
<html><head><title>Hello1</title></head><body>
<span>Hello, </span>
<xsl:value-of select="/pdata/person[1]/name/title" />
<xsl:text> </xsl:text>
<xsl:value-of select="/pdata/person[1]/name/last" />.
</body></html>
</xsl:template>
```

You will notice that some elements in this code have tag names that start with `xsl:`. These are instructions to the XSLT processor. The first of them, `<xsl:template>`, establishes the context in which its content is instantiated. The context is specified by the `match` attribute whose value is an XPath expression.

The content of our `xsl:template` element consists of an HTML page with occasional insertions from our XML data file. We extract the data using the `xsl:value-of` element. The `select` attribute, whose value is an XPath expression, indicates the data to be extracted. To output whitespace between "Mr." and "Monster", we use `xsl:text`.

All elements whose tag name does not start with `xsl:` are passed through to output without change. We end up with:

```
<html><head><title>Hello1</title></head><body>
<span>Hello, </span>Mr. Monster.
</body></html>
```

An Improvement with `xsl:variable`

We will do a lot more XSLT and XPath in the chapters to come, but we'll present a few details here, so you can do simple experiments. Listing 1-11 repeats the same XPath expression (`/pdata/person[1]/name/`) twice. We can remove that inefficiency by using `xsl:variable`, as in the highlighted lines of Listing 1-12.

Listing 1-12. Same, Using an XSLT Variable

```
<xsl:template match="/" >
<html><head><title>Hello2</title></head><body>
<span>Hello, </span>
<xsl:variable name="nm" select="/pdata/person[1]/name" />
<xsl:value-of select="$nm/title" />
<xsl:text> </xsl:text>
<xsl:value-of select="$nm/last" />.
</body></html>
</xsl:template>
```

As we explained in the section on XPath expressions, we'll get the same result if, instead of using `/pdata/person[1]`, we use `/pdata/person[@id='CM123']`. In fact, in this case we'll get the same result if we simply use `/pdata/person` because there is only one person on the selected list of nodes.

Processing a Set of Nodes

How do we select and work with more than one node? Selecting is easy: the expression `/pdata/person` selects all the person nodes that are children of `pdata` nodes (of which there is only one) that are children of the root. To work with each of them, we use `xsl:for-each`. Let's say hello to every person in our file, as shown in Listing 1-13.

Listing 1-13. Working with Multiple Nodes

```
<xsl:template match="/" >
<html><head><title>Hello Everybody</title></head><body>
<xsl:variable name="plist" select="/pdata/person" />
<xsl:for-each select="$plist">
  <span>Hello, </span>
  <xsl:value-of select="name/title" />
  <xsl:text> </xsl:text>
  <xsl:value-of select="name/last" />
  <br/>
</xsl:for-each>
</body></html>
</xsl:template>
```

Note that the select expressions in the body of `xsl:for-each` use relative path expressions that do not begin with a "`/`". This is quite similar to how we refer to directory paths.

Conditional Processing

Suppose some people don't have a title listed because they prefer to be called by their first name. As we go through the list, we want to make our output conditional on whether the title is present. The highlighted code in Listing 1-14 shows how we do it.

Listing 1-14. An XSLT Program with Conditional Expressions

```
<xsl:template match="/" >
<html><head><title>Hello Everybody</title></head><body>
<xsl:variable name="plist" select="/pdata/person" />
<xsl:for-each select="$plist">
  <span>Hello, </span>
      <xsl:variable name="title" select="name/title" />
      <xsl:choose>
      <xsl:when test="$title"> <!-- true if non-empty -->
        <xsl:value-of select="$title"/><!-- same output as before -->
        <xsl:text> </xsl:text>
        <xsl:value-of select="name/last" />
      </xsl:when>
      <xsl:otherwise><!-- output just the first name -->
        <xsl:value-of select="name/first" />
      </xsl:otherwise>
    </xsl:choose>
    <br/>
</xsl:for-each>
</body></html>
</xsl:template>
```

This should be a fairly understandable (if a bit verbose) way of saying if-then-else.

Outputting Attributes with Computed Values

Suppose we want to format the output so the name of the favorite drink appears in the favorite color. The recommended way to do this in HTML is by using a style attribute, as in

```
<span style="color:red">cookie juice</span>
```

The obvious (but wrong) thing to do would be to insert the desired HTML into the XSLT stylesheet as we have done in the preceding examples. However, in this case, we have to place an XSLT expression inside the attribute value:

```
<span style="color:<xsl:value-of select="favorites/color"/>">
```

This violates several rules of XML syntax (as we explain in the next chapter), but, even if we fix the syntax, the basic problem is that our XSLT expression is in quotes and will remain unevaluated by the XSLT processor. This problem has two solutions: one using more XSLT and the other using a special notation to evaluate XSLT within attributes. In either case, we start by creating an XSLT variable, `styleOut`, whose value will be the string `"color:red"`:

```
<xsl:variable name="styleOut">
  <xsl:text>color:</xsl:text>
  <xsl:value-of select="favorites/color"/>
</xsl:variable>
```

One way to use this would be within an `xsl:attribute` element that constructs an output attribute:

```
We have yummy
<xsl:element name="span">
  <xsl:attribute name="style"><xsl:value-of select="$styleOut"/></xsl:attribute>
  cookie juice
</xsl:element>
```

Alternatively, we can simply enter the output HTML in the XSLT stylesheet and use special notation for XSLT expressions within attribute values to get them evaluated. The special notation is curly brackets placed around XSLT variable that needs to be evaluated:

```
We have yummy
<span style="{$styleOut}">
  <xsl:value-of select="favorites/drink"/>
</span>
```

The Framework Around the Code

The XML framework in which Listings 1-11 to 1-14 operate is shown in Listing 1-15. Its first line is the XML declaration because an XSLT program is an XML document. Its second line is the start tag of the root element whose tag name is `xsl:stylesheet`. Its attributes declare the XSLT namespace and specify the current version.

Listing 1-15. The Outer Shell of an XSLT Program

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
<!-- The programmatic core is inserted here -->
</xsl:template>
</xsl:stylesheet>
```

The notion of an XML namespace is not an easy one, and we will spend quite a bit of time on it in the next chapter. For now, just note that the namespace declaration associates a prefix (xsl) with a unique URL. In the rest of the program, the prefix indicates that the name after the prefix (such as `template` or `value-of`) belongs to the namespace identified by that URL. In this case, this serves as a signal to the XSLT processor that the element with the prefix-qualified name is an XSLT instruction. (In general, the "meaning" of a namespace is completely determined by the application that receives XML data from the parser.) Although the xsl prefix is traditionally used, it is, in fact, arbitrary; any prefix would do as long as it is associated with the right namespace.

Stylesheet Languages and Browsers

We have shown how XSLT can be run on the server to produce HTML to be displayed in the browser. There are other ways to apply an XSLT program to XML data, and other ways to display XML in the browser. In this section, we will try to sort out the messy relationships among browsers, stylesheet languages, and XML.

To display an XML document in the browser, it needs to be associated with a stylesheet. If you don't provide a stylesheet of your own and open the document in the browser as a local file, it will be displayed using a default stylesheet that preserves the markup and shows the tree structure of the document. If you want to provide a stylesheet of your own, you can choose from three stylesheet languages: CSS, XSLT, and XSL-FO (working individually or in combinations). To understand and evaluate the possibilities, we need to clarify what those languages are and how well they are supported.

Stylesheet Languages: A Brief History

Cascading Stylesheets (CSS) is the oldest stylesheet language for the Web. First developed for use with HTML, it is a very simple, intuitive language with somewhat limited capabilities. A CSS stylesheet consists of a set of rules. Each rule consists of a selector and a declaration in curly brackets. The simplest selector is

a tag name, but a selector can also be a list of tag names or a tag name in context. A declaration is a set of property-value pairs separated by semicolons, and within each pair the property is separated from its value by a colon. For instance, the following rule centers `<p>` elements in the page, gives them a solid brown border three pixels wide, and sets the background color to blue:

```
p {
  margin-left:5%;  width:90%;
  border: solid 3px brown;
  background-color:blue
}
```

That's the essence of CSS; the rest is a multitude of detail. In particular, you need to know the following details:

- the syntax of selectors
- names of properties and their possible values
- absolute and relative units of measurement
- color representations
- simple rules of inheritance

All of these can be found at `www.w3.org/Style/CSS/` and multiple excellent tutorials on the Web and in print. All the major browsers (Internet Explorer, Netscape, Mozilla, and Opera) support CSS for HTML quite well.

CSS for XML

CSS can also be used with XML. Consider the simple schematic XML document shown in Listing 1-16 (in which `r` stands for root, and `c` stands for child):

Listing 1-16. Simple XML Document, xml4css.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="style0.css" type="text/css"?>
<r>
  <c a1="attr1" a2="attr2"> First child, two attributes</c>
  <c>Second child, no attributes</c>
  <d>A deviant child</d>
</r>
```

The second line of this document (technically a processing instruction) associates a CSS stylesheet with it. The stylesheet looks like Listing 1-17.

Listing 1-17. Simple CSS Stylesheet, style0.css

```
r {background-color:#ffffef}
c, d { display:block}
c {padding:5;margin:5;
    border:solid 3px;
    width:200;
    text-align:center;
    font-family:verdana;
    font-size:14;
    color:maroon
}
d {background-color:lightblue;
    margin-left:50;
    border:solid 3px;
    width:200;
    text-align:center;
    font-family:verdana;
    font-size:36;
    color:green
}
```

If you place xml4css.xml and style0.css in the same folder and open the XML file in Internet Explore 6 or another XML- and CSS-aware browser, the result will look like Figure 1-7.

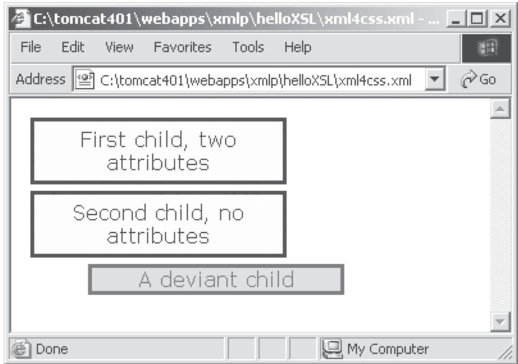

Figure 1-7. XML with CSS in the browser

XSL, XSLT, and XSL-FO

For all its simplicity and effectiveness, CSS has the following limitations:

- CSS syntax is not XML and so requires a different parser.
- CSS lacks facilities for selecting, sorting, and otherwise rearranging data.
- CSS lacks high-end layout capabilities, such as multiple column layout, footnote placement, and conditional formatting.

When XML was invented, work immediately started on a new stylesheet language to accompany XML. Initially, there was a single XSL language whose design closely followed DSSSL, the stylesheet language for SGML created by James Clark (who was also the technical lead in creating XML). The goal was to produce a stylesheet language for specifying how an XML document is to be displayed in multiple media, including the Web browser. Although that goal has not yet been reached, two other specifications—XSLT and XPath (both having to do with transforming rather than formatting XML)—became W3C recommendations in November 1999, even though they did not exist as independent projects until fairly late in that year. As the XSL project unfolded, different parts of it grew at different speeds, and their relative importance and state of preparedness were changing. Eventually, XSLT and XPath were carved out into separate projects and completed, whereas the document formatting part spent another two years before it became Recommendation in October 2001.

That a style sheet language was needed for XML to function was obvious from the beginning: if users can define their own elements, they have to be able to specify how those elements will look when displayed in the browser window or other media. Also, from the beginning, the intent was to give XSL the ability to add, remove, and reorder the elements of the document tree, so that, for instance, the stylesheet could handle multiple reports from a database table, showing different fields and sorting records in different ways. As part of this functionality, XSL needed a way of referring to nodes and sets of nodes in the tree to select them for processing.

Initially, the tree-transformation part of XSL was just an aid to the formatting part, but it proved to be easier to develop and build a consensus about. As XML's role was evolving from a tool for document markup to (also being) a tool for data interchange among applications and components of applications, the transformation "module" was developing an independent significance, totally unrelated to formatting and display. At some point, a single XSL split into XSL for formatting and XSL for transformation (XSLT). The XSLT part was taken over by James Clark who brought it to a swift completion while at the same time producing xt, an open-source reference implementation of the XSLT processor.

> **NOTE** *James Clark has since discontinued support for XT. It is supported and further developed by* `http://4xt.org`.

As XSLT was taking shape, it was realized that the ability to select sets of nodes in a systematic way was needed not only for tree transformations but also for linking. As a result, that project also developed an independent existence under the name of XPath, and it was completed (on the same date as XSLT) by James Clark and Steve DeRose.

The use of XSLT spread rapidly. Several excellent implementations are now available from Microsoft, Apache (based on work initially done at Lotus and Sun), Oracle, and Michael Kay. Of all the XML technologies that have been developed since 1997, XSLT and XPath are unquestionably the most successful and important.

XSL-FO

The formatting part of XSL took much longer to develop: it became a W3C recommendation only in October 2001. Its target area of application is high-end publishing, not necessarily in the browser, and not even necessarily in electronic form. To quote the specification: "Given a class of arbitrarily structured XML documents or data files, designers use an XSL stylesheet to express their intentions about how that structured content should be presented; that is, how the source content should be styled, laid out, and paginated onto some presentation medium, such as a window in a Web browser or a handheld device, or a set of physical pages in a catalog, report, pamphlet, or book" (`www.w3.org/TR/xsl/slice1.html#section-N629-Introduction-and-Overview`).

To use XSL-FO, you need, in effect, two programs: an XSLT processor that converts XML to be displayed into an XSL-FO document and a rendering engine. In practice, the XSLT program operating as part of XSL-FO will typically perform two operations: a "pure" transformation to create the desired view of XML data (filter, rearrange, and add content as needed) and the XSL-FO transformation, to produce the desired display description. In this book, we concentrate on pure transformations, but we will show how to arrange multiple XSLT programs in a processing pipeline.

Displaying XML on the Client

With the relationships among formatting languages clarified, we can list possible ways to display XML in the browser. Background information follows Table 1-2.

Table 1-2. Approaches to Displaying XML in the Browser

APPROACH	BROWSER REQUIREMENTS
Display XML directly, using CSS	XML- and CSS-aware browser
Transform XML into (X)HTML on the client using XSLT and display	XML- and XSLT-aware browser
Transform XML into (X)HTML on the server using XSLT or programmatic APIs and display	Any HTML browser
Use XSLT to transform XML into an XSL-FO document on the server; further transform into a displayable format such as PDF	Any browser with a PDF plugin

XML+CSS

The latest versions of Internet Explorer (IE), Netscape, Mozilla, and Opera implement XML parsing and the basics of CSS styling quite well; so, for very simple cases, the first approach is workable. Opera and Mozilla are pushing this approach further by implementing CSS2 support for XML (in addition to HTML). With CSS2, one can display bulleted lists and tables directly in XML without transforming it into HTML. However, IE lags in this respect, and there are reports that Microsoft does not consider XML+CSS an important feature to support (`http://lists.xml.org/archives/xml-dev/200109/msg00194.html`).

XML+XSLT on the Client, and More on Processing Pipelines

The idea is to be able to give the browser the URL of an XML document that contains another URL for the stylesheet (which is also an XML document), and the browser displays the output of the transformation:

```
<?xml version="1.0"?>
<?xml-stylesheet href="URL for stylesheet"  type="text/xsl"?>
<!-- the rest of the XML document -->
```

Consider the steps involved in this processing chain:

- As the document is loaded as a stream of characters, the XML parser within the browser parses it into a tree and identifies the processing instruction for the stylesheet.
- The stylesheet is also loaded as a stream of characters and parsed into a tree.
- The XSLT processor within the browser applies the stylesheet to the input tree; the browser renders the result using either a default stylesheet for HTML or a stylesheet for XML.

The salient features of the process are as follows.

- Both the input document and the stylesheet can be anywhere on the Internet.
- The software involved—the XML parser and the XSLT processor—is standard and free.
- The output of the process can be either displayed in the browser, piped into another processor, or both.

At this point, we would like to recapitulate, with more background than before, the reasons why XML has been so quickly and widely accepted.

- It is easy to switch between the character-sequence view of XML and the tree-structured-data view of XML. The software to perform that switch (the XML parser) is standard, of high quality, ubiquitous, and free.
- Character sequences are easy to send over the network using standard protocols.
- Tree-structured data is easy to work with. In particular, it is easy to transform one tree into another, changing either the text content, the markup, or both. The software that performs these tasks (the XSLT processor) is standard, of high quality, ubiquitous, and free. The APIs for working with XML tree data are open and widely supported standards.
- Because XML can encode both data and metadata, applications that communicate using XML can discover each other and establish a communication channel without prior arrangements.

- It is easy to construct pipelines of XML processors in which each processor receives XML data, does some transformation and/or computation on it, and sends the result as XML to the next processor. (See Figure 1-8.)

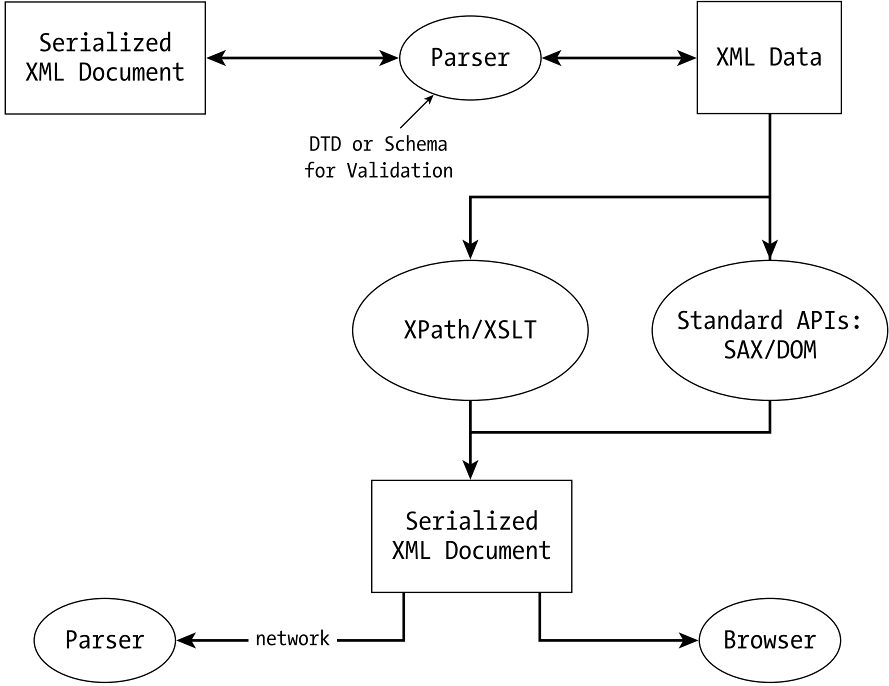

Figure 1-8. XML processing pipeline

Four adjectives are repeated in this list of features: *easy, standard, ubiquitous,* and *free*. These are the key to XML's success. XML is based on several simple ideas involving free software and wide acceptance. They combine to make XML a major enabling technology for interoperable distributed applications, themselves an enabling technology for commerce and cooperation.

Browser Support

XSLT support on the client side is, as of this writing, available for IE on the Windows platform, Mozilla, and Netscape. Mozilla and Netscape offer less support, but continue to improve. IE support follows a complex trajectory that deserves a separate section.

XSLT, Internet Explorer, and MSXML

Microsoft released a browser with XSLT support (IE5) before XSLT was finalized as a W3C recommendation. IE5, with MSXML 2 as XML parser and XSLT processor, supported a working draft of XSLT that was very different from the eventual standard. This resulted in great confusion among IE5 users and a great deal of indignation among the experts who had to explain in many forums why XSL files that work perfectly well in Internet Explorer do not work properly with other processors.

MSXML 3 has implemented a fully conformant version of both the XML parser and XSLT processor. Although the browser continued to ship with the old nonconformant version, it is possible to download MSXSML 3 and install it with IE5.5. (See Appendix A for installation instructions.) The result is that you can indeed test XSLT programs on XML data by simply opening an XML file in the browser (assuming the file has a link to an XSLT stylesheet).

MSXML 4 further improves both the parser and the XSLT processor, but, somewhat paradoxically, IE6 does not ship with MSXML 4. It does ship with the fully-conformant MSXML3. As for MSXML 4, it cannot be used with a browser at all, except via a script that creates the appropriate object and calls its methods. (You will see examples in later chapters.) In summary, XSLT support in major browsers (IE, Netscape, Mozilla) continues to improve. The strategy of transforming XML into XHTML for display in the browser may soon become a viable option, if you don't mind additional computational load on your client machine.

XML+XSLT => HTML on the Server

This approach puts minimal requirements on the client and also reduces its computational load. Its drawback is that the client receives a document stripped of its metadata tags, which are replaced by the generic tags of HTML: even in a well-designed HTML or XHTML document, it is not easy to distinguish between a table of books and ISBNs and a table of last names and email addresses.

XML+XSLT => XSL-FO on the Server

This approach, as we mentioned, is not quite ready for prime time: the XSL-FO specification has been released as a W3C recommendation very recently (October 17, 2001), and rendering engines for browsers are not even mentioned in the list of features for the next release. (However, it is possible to transform XSL-FO into PDF and display using an Adobe plugin.) In general, XSL-FO intends to compete with professional formatting languages and tools such as TeX, QuarkXPress, and FrameMaker. For Web page display in the browser, CSS in combination with XSLT is completely adequate.

Conclusion

In this chapter, we have covered the very basics of XML: what it is, how it evolves, who is in charge, and why it is great. The key concepts we introduced involve language, markup language, syntax (grammar), parsing, and interpretation. We explained how an XML parser converts a well-formed XML document into a tree structure. We presented the basics of XSLT. In the end, we had enough background to make the case that XML is the key technology of the Internet because it enables interoperability among programs and cooperation among people and organizations.

CHAPTER 2

Well-Formed Documents and Namespaces

WITH BASIC DEFINITIONS and examples behind us, we can move on to a detailed discussion of the specifications. In this chapter, we concentrate on documents without DTDs because they have a simpler structure. Although occasionally mentioned in this chapter, DTDs and other approaches to validation (such as XML Schema and RELAX NG) will be introduced in Chapter 3.

In outline, this chapter proceeds as follows:

- HTML vs. XHTML
- XHTML modularization and XHTML Basic
- well-formed XML documents
- names and namespaces
- global attributes and XLink
- namespace URI and RDDL (XHTML Basic + XLink)

We will start with a comparison of HTML and XHTML.

HTML, XML, and XHTML

HTML is by far the most familiar markup language. We will review its main features in comparison with XHTML to emphasize, one last time, the following basic facts.

- HTML is a specific language defined in the SGML framework.
- XML is not a language but a framework for defining languages.
- XML is a revision of SGML.

The main difference between XML languages and HTML and other SGML languages is that XML documents can be parsed without a DTD, whereas SGML documents (whether in HTML or any other SGML language) can be parsed only with the help of the DTD. This is because, in SGML languages, the end tag of an element can frequently be omitted even if the element is not empty: in HTML, you don't have to close off your `<p>`s with a `</p>`. For HTML empty elements, the end tag is always optional: nobody puts `<br></br>` in a Web page.

HTML vs. XHTML

Listing 2-1 provides an example of a perfectly grammatical HTML document (paralist.htm); it uses CSS within a style attribute to specify the font properties for the first `<p>` element:

Listing 2-1. An HTML Document

```
<html>
<head><title>HTML Example</title></head>
<body bgcolor="#ffffef">
  <h1>Heading</h1>
  <p style="color:maroon;font-size:2em">a paragraph with <em>italics</em>
 followed by a list
  <ul>
    <li>item one
    <li>item two
  </ul>
  <p>Another paragraph with a line break <br> in the middle.
</body>
</html>
```

What would the element tree for this document look like? Figure 2-1 shows one possibility.

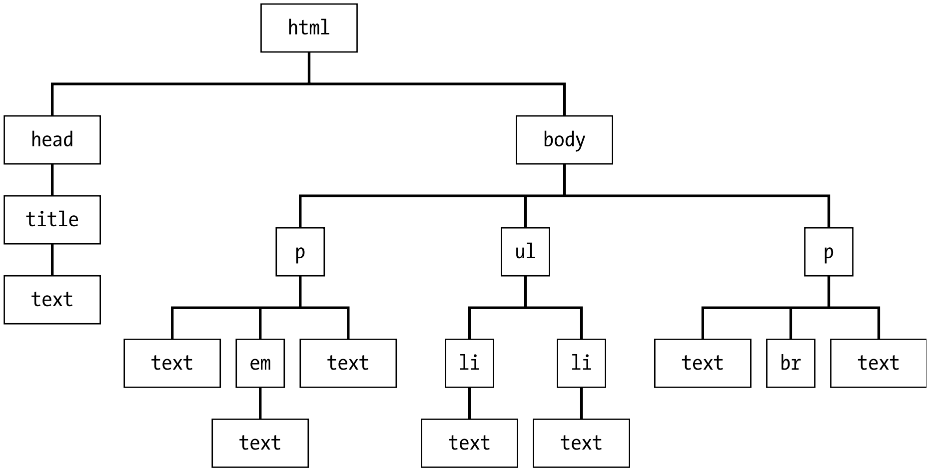

Figure 2-1. Element tree of an HTML document

Is this the only possible tree? Note that the <p> elements don't have an end tag, so it would be consistent with the markup to make the <ul> element a child of the first <p>. In fact, we could even make the second <p> a child of the first. Is there a "correct" structure among these possibilities? The question is not academic because the page uses CSS, and a CSS style defined on an element is inherited by the element's children. If <ul> is a child of <p>, its font will be large and maroon; otherwise, it will be small and black. Obviously, we can't leave this decision to the browser's parser: we need a rule. There is, indeed, such a rule; in fact, for every HTML element, there is a rule that stipulates which elements it can contain. The rule for <p> lists many possible children, but <ul> is not among them. As Figure 2-2 shows, the browser knows and obeys the rule.

Figure 2-2. HTML document in the browser

The rules of HTML are stated in the HTML DTD, which is part of the W3C HTML recommendation. (The latest and final version is 4.01.) HTML DTDs are very similar to XML DTDs, and we are not going to discuss their minor differences. The essential point is that, to process an HTML document and build its tree, the browser's parser needs to know the grammar of HTML (the HTML DTD). The corresponding XML document terminates each element with an end tag and can be parsed without a grammar. (Empty elements consist of a single tag terminated with the "/>" sequence, as in `<br />`.) Modified in this way, HTML becomes XHTML, officially described by W3C as "a Reformulation of HTML 4 in XML 1.0".

To parse the document shown in Listing 2-2, the browser wouldn't need the grammar anymore.

Listing 2-2. An XHTML Document

```
<html>
<head><title>HTML Example</title></head>
<body bgcolor="#ffffef">
  <h1>Heading</h1>
  <p style="color:maroon;font-size:2em">a paragraph with <em>italics</em>
 followed by a list</p>
  <ul>
    <li>item one</li>
    <li>item two</li>
  </ul>
  <p>Another paragraph with a line break <br/> in the middle.</p>
</body>
</html>
```

HTML As a Language

In Chapter 1, we defined a language as consisting of a vocabulary and a grammar. Is HTML a language in the sense of this general definition? It most certainly is because it has a fixed vocabulary of element and attribute names whose usage is controlled by specific grammar rules. Is HTML a formal language or an interpreted one? This question is a little trickier: HTML itself does not define the meaning of its expressions, but they are always given a meaning in a stylesheet (either the default stylesheet that comes with the browser or a custom stylesheet supplied by the user). For instance, the meaning of the `<h1>` tag in the following line is something like: "display the text 'Heading' in a large font, bold face" (for example, 24 point Times New Roman).

```
<h1>Heading </h1>
```

As an HTML author who is also well versed in CSS, you can change that meaning by redefining the style as in the following line of code.

```
<h1 style="font-family:algerian;color:red;font-size:4em">Heading</h1>
```

Figure 2-3 shows the resulting document, paralist2.htm, in the browser.

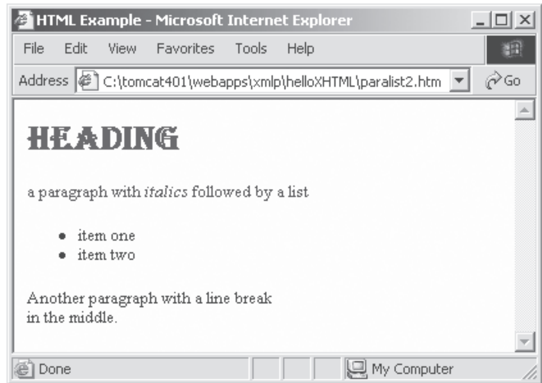

Figure 2-3. HTML Document with a different stylesheet

However, your creativity has limits: all possible meanings are about how the text within the tags is to be displayed in the browser. This is what HTML expressions within the body are mostly about. The important point here is that the meaning of HTML tags and attributes is determined by a stylesheet and a specific application—the browser—that interprets it.

By contrast, XML is not a language. It doesn't have a specific markup vocabulary or a specific grammar. Instead, it has a DTD language for defining vocabularies and grammars. In addition, XML makes no assumptions at all about the meaning of its markup languages. They can be intended for any application, and the application alone determines the meaning of the markup.

SGML/HTML = XML/XHTML

In a sense, it is unfair to compare HTML to XML: it's like comparing a cookie to a cookie cutter. Or, to change the metaphor, XML is not HTML's sibling but more like its (youthful and vigorous) uncle. Figure 2-4 illustrates the relationship.

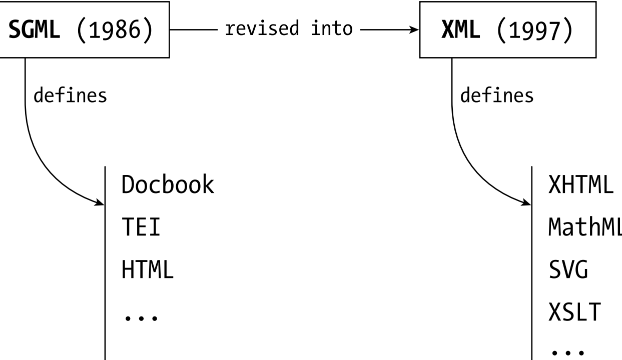

Figure 2-4. SGML and XML

A good comparison would be between HTML and XHTML. More precisely, we want to compare an HTML document with the corresponding XHTML document; Listing 2-1 and 2-2 give us material for comparison. The most important difference between them, from which more-specific differences follow, is that an XHTML document can be checked for well-formedness and parsed in the absence of a grammar. We have expressed this difference by saying that elements must form a tree in all SGML languages as well as in all XML languages, but, in XML languages, additionally, the document's markup must explicitly show the tree structure.

HTML and XHTML have several other, less important differences, such as XML is case sensitive and XHTML tags are defined to be in lowercase. (A good place to look for a summary of all such differences is Section 4 of the XHTML spec that lists its differences with HTML 4.0, `www.w3.org/TR/xhtml1/#diffs`.) However, most remaining differences between XHTML and HTML pages result from the behavior of the parser that processes them rather than from any differences in their grammar and underlying framework.

Parsers with Attitude

HTML parsers are famously tolerant of ungrammatical Web pages: they will display a page without complaint even if it lacks the `<html>` and `<head>` elements, has unquoted attribute values, and shows other violations of HTML rules. For instance, the page shown here (paralistbad.htm) will display correctly in both IE and Netscape browsers, possibly with differences in whitespace:

```
<p style=color:maroon;font-size:2em>a paragraph followed by a list
  <li>item one
  <li>item two
```

As you can imagine, this means a lot of work both for parser writers, and for parsers themselves. All SGML parsers are complex, but HTML parsers are even more so because they try to anticipate and correct users' mistakes. XML is more economical on all counts. (One of XML's design goals was that it shall be easy to write programs that process XML documents.) XML parsers, especially nonvalidating ones, are small and relatively easy to write both because XML is simpler than SGML and because XML's attitude to syntax errors is totally negative.

Error handling by XML parsers is not only strict, it is also uniform. The W3C XML recommendation, in a section on conformant processors, precisely specifies what a processor must do in response to different kinds of errors. There is, as we mentioned in Chapter 1, a test suite that is designed to test the parser's compliance with the XML specification, especially in its error handling. (See `http://oasis-open.org/committees/xml-conformance/xml-test-suite.shtml`.)

There are two main reasons for this strict attitude. First, XML parsers are frequently used to mediate between computer applications or components within an application. XML data is often generated by a program and consumed by another program that performs computations on it. In this sort of configuration, ill-formed data must be inadmissible. (In particular, it must be inadmissible to feed the same XML data into two different browsers and see one of them succeed and the other fail to parse it.) Second, XML was developed from the start in anticipation of small mobile devices. A parser sitting in a cell phone, wristwatch, or remote sensor cannot afford the megabytes of memory that are needed to anticipate and accommodate grammatical error.

> **NOTE** *Dave Raggett, a longtime staff member at W3C, wrote a remarkable program called Tidy (*`http://www.w3.org/People/Raggett/tidy/`*). It performs several functions on HTML documents: fixes grammatical errors, points out deprecated features, and converts the HTML document to XHTML. We will use Tidy in Chapter 7.*

Why XHTML?

Why use XHTML instead of HTML? The main reason is that the entire array of XML technologies becomes available to you. If you want your Web page to be produced by an XSLT program, you have to make your template HTML material conformant to XML rules because an XSLT program is an XML document, and if you enter, for example, `<br>` instead of `<br />`, the parser will object.

Since 1999, HTML has been in effect mothballed while XHTML has been an area of active development. A quick look at the list of W3C recommendations shows *XHTML 1.1—Module-Based XHTML* (May 2001) and *XHTML Basic for*

Small Devices (December 2000). If you look inside *MathML 2.0* (February 2001), you will see that "it is designed to be used as a *module* in documents marked up with the XHTML family of markup languages" (Appendix A2). Outside W3C, XHTML has been used to define RDDL (Resource Directory Description Language), a promising new idea that we will introduce in the section on namespaces.

> **NOTE** *All new XHTML-based or XHTML-related languages rely on XHTML modularization, a framework for dividing the large vocabulary of HTML into small modules that can be independently reused in various ways. We present XHTML modularization in Chapter 3.*

XML Documents Without a DTD

XML documents have logical and physical structure. Logically, a document consists of elements, attributes, and other less important items, such as comments. Some types of items are processed away in parsing, and the rest are represented as a tree of nodes. The *Infoset* recommendation regulates what gets preserved in the tree.

Physically, a document is a unit of storage (such as a file or a string) for character data and markup that can include other such units by reference. A generic name for "units of storage" is "entity." Most entities have to be declared in a DTD before they can be used, but two groups of entities can appear in documents without a DTD. We explain about entities and CDATA sections before presenting a complete summary and outline of an XML document without a DTD.

Character Entities and Five Predeclared Entities

Character entities represent individual characters by their Unicode numbers, either decimal or hexadecimal. They are used for markup characters and characters that are difficult to enter from the keyboard. To refer to a character entity within a document, place it between an ampersand and a semicolon. For instance, the copyright symbol (©), whose Unicode number is x00A9, can be entered into your document in three ways:

```
&#169; &#xA9; &#x00A9;
```

The ampersand itself can also be entered as a character reference (&), but it has a special name in addition to the numeric code. Such special names are

predeclared for five characters. (See Table 2-1.) To refer to a character by its special predeclared name, place it between an ampersand and a semicolon.

Table 2-1. Five Predeclared Entities

CHARACTER	ENTITY NAME	REFERENCE	DECIMAL CODE	HEX CODE
&	amp	&	&	&
<	lt	<	<	<
>	gt	>	>	>
"	quot	"	"	"
'	apos	'	'	'

CDATA Sections

If you have a section in your document that contains a great number of markup characters (such as XML source or Java code), you may want to enter the entire section as a CDATA section that is not parsed by the processor. The syntax is as follows,

```
<![CDATA["<" & ">" are "angle brackets"]]>
```

Within XHTML, it is common to put the contents of `<script>` and `<style>` elements in CDATA sections.

The markup of CDATA sections is removed in parsing. The boundaries of CDATA sections leave no trace in the XPath tree model but may be preserved in DOM. They are not preserved in the document's infoset, and the next version of DOM will conform to the *Infoset* recommendation and align itself with XPath.

Summary, Outline, and EBNF Productions

In summary, documents without a DTD can contain the following kinds of material,

- Declaration: Optional but highly recommended, as in

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

- Elements and attributes: This is the informational core of the document.
- Comments:

```
<!-- this is a comment -->
```

- Processing instructions (PIs) (in XML, they are rarely used except to provide a stylesheet reference):

```
<?xml-stylesheet href="style0.css" type="text/css"?>
```

- Character entity references.
- References to five predeclared entities: `lt`, `gt`, `quot`, `apos`, and `amp`.
- CDATA sections: A document without a DTD follows this outline.

```
<![CDATA[if(a<b && b<c) return;]]>.
```

- XML declaration: Nothing can precede it, not even a comment or whitespace.
- Miscellaneous optional material: Comments and PIs, whitespace as desired. (In a document with a DTD, the DTD or a DTD reference would appear here.)
- The start tag of the root element.
- All other material, well formed.
- The end tag of the root element.

It is actually easier to say this in EBNF (Extended Backus-Naur Form) than in English. EBNF is a formal notation for describing the syntax of programming languages, and it is also used in many XML specifications. In *XML 1.0*, it is used to define the well-formedness and validity rules of XML documents. The entire *XML 1.0* boils down to 89 EBNF rules, or "productions" as they are called. It is useful to be able to read EBNF productions because they pack a lot of information in very few lines of text. Here is a small sample, numbered as in *XML 1.0*, with brief comments. As you read the rules, remember that the characters ?, +, and * indicate the number of repetitions: ? stands for 0 or 1, + stands for 1 or more, and * stands for 0 or more.

Production 1 specifies the structure of a document:

```
[1]  document ::=  prolog element Misc*
```

This says that a document is composed of a prolog, followed by a single element (the root of the element tree), followed by optional miscellaneous material.

Prolog is composed of optional elements, as follows: an optional XML declaration, followed by Misc*, followed by an optional DTD, again with Misc* thrown in:

```
[22]  prolog ::=  XMLDecl? Misc* (doctypedecl Misc*)?
```

Misc* is any sequence of whitespace, comments, and PIs. Here's the Misc production (S stands for whitespace, and the vertical bar means "OR"):

```
[27]  Misc ::=  Comment | PI |  S
[3]    S   ::=  (#x20 | #x9 | #xD | #xA)+
```

We will show more EBNF in the namespace section later in the chapter.

A "Kitchen Sink" Example

Listing 2-3 shows everything you can find in a DTD-less document (ksink.xml). Note that, to display escape sequences in the browser, we have to escape the escapes: for instance, to display < we have to enter <. Because CDATA sections cannot be nested, we have to enter its closing character sequence outside the section itself, using entity references. The document is followed by a modest CSS stylesheet (ksink.css, Listing 2-4) and a screenshot (Figure 2-5).

Listing 2-3. All You Can Find in an XML Document Without a DTD

```
<?xml version='1.0' encoding='utf-8'?>
<!-- The line above is the XML declaration. Nothing can precede it,
      not even comments or whitespace.
     The line below is a PI. It associates a stylesheet with the document.
     In XML, PIs are rarely used for any other purpose.
-->
<?xml-stylesheet href="ksink.css" type="text/css"?>
<root_elt>
<h1>Document without a DTD</h1>
<non_empty_element>
This is element content, parsed character data (PCDATA). <br />
To insert a markup character here, such as <, you have to use a reference
```

```
to a <em>pre-declared general entity</em>, &amp;amp;lt;
or a <em>character entity</em>, &amp;amp;#60;. <br />
If you have many such characters, you can put them all into a
<![CDATA[CDATA section: <[CDATA[ (a<b && b>c) ]]>. ]]&gt;.
CDATA sections cannot be nested. <br />
You can also use character entities to insert characters that are not easy
to enter from the keyboard, such as the copyright character &#xA9;.
(In this case, we used the character's hexadecimal code, &#38;#38;#xA9;.) <br />
</non_empty_element>
</root_elt>
```

Listing 2-4. A Minimalist Stylesheet

```
rootElement, h1, non_empty_element, br {
  display: block; margin-bottom: .6em;
}
h1 {font-weight:bold;font-size:large;text-align:center;}
em {font-weight:bold;font-style:italic}
```

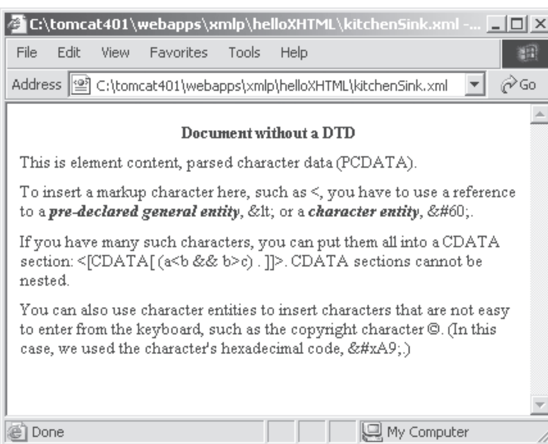

Figure 2-5. XML document without a DTD

Names and Namespaces

A markup language is a vocabulary of names: the names of elements and attributes. The names of attributes must be unique within an element, but different elements can have attributes of the same name. We can describe this by saying that attributes of different elements belong in different namespaces. The notion of a namespace is very familiar from programming: an object in C++ or Java is a namespace for its variables and methods; a package in Java is a namespace for

its class names; a database is a namespace for the names of its tables; and a database table is a namespace for the names of its fields. The ability to partition the names in your program or database into different namespaces is essential for preventing name conflicts.

The initial *XML 1.0* specification has no provisions for partitioning element names within a document into different namespaces. This could potentially result in name conflicts when two different vocabularies are merged in a single document. The danger is real for XML languages—such as XSLT, SVG, XML Schema, or JSP—that are designed for use with a great many other languages. It would be reckless to leave the vocabularies of such languages unprotected. *XML Namespaces* (1998) was primarily developed to protect the vocabularies of such widely used XML languages.

Namespaces and Prefixes

A common way to create a globally unique name is by forming a pair that consists of a namespace prefix and a local name. The namespace prefix uniquely identifies the namespace, the local name must be unique within that namespace, and the combination of the two creates a globally unique name. This is how Java classes and packages operate: the name of a package is globally unique (or at least has a good chance of being so), class names are unique within a package, and a fully qualified class name consists of the package name as a prefix, followed by a period and the class name. Reversed URLs are often used as package names: a good deal of Sun's software is in the com.sun package or its subpackages, for instance:

```
com.sun.xml.parser.Resolver res = new com.sun.xml.parser.Resolver();
```

Here, `Resolver` is a local name, `com.sun.xml.parser` is both the name of the package and a unique namespace prefix, and the combination of the two is a fully qualified, globally unique name of Sun's Resolver class.

The designers of *XML Namespaces* had a well-known source of globally unique names ready at hand: the URL, or its generalization, the URI. It was a natural decision to make it a source of unique namespace prefixes, so that a unique element name would consist of a URI prefix to identify the namespace and a local name that is unique within that namespace. Conceptually, if our company URL is `http://www.n-topus.com`, and we want to put our Address element in a protected namespace, we would say that its fully qualified globally unique name is something like `{http://www.n-topus.com/elemnames}Address`.

The problem is that this name, as written, is not a legal XML name, and it's also extremely long. The solution of *XML Namespaces* is to use a two-step procedure for establishing a namespace. In the first step, a unique namespace URI is

declared and mapped to a prefix that contains only legal characters. In the scope of that declaration, the prefix serves as a proxy for the namespace URI. Here is an example that you have already seen as Listing 1-15:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
<!-- The rest of the program goes here -->
</xsl:template>
</xsl:stylesheet>
```

The second line of this code contains a namespace declaration that maps the URI to a prefix. (The prefix for this particular namespace is usually `xsl`, but any prefix would do as long as it is associated with the right URI.) Syntactically, the declaration is an attribute. The name of the attribute consists of a reserved sequence of characters, `xmlns`, followed by a colon and the prefix to which the namespace URI is mapped; the value of the attribute is the namespace URI. The scope of the declaration includes the element whose start tag contains that declaration and all the descendants of that element (unless there is another declaration with more local scope, as discussed shortly).

The XML 1.0 *Perspective vs. the* XML Namespaces *Perspective*

A namespace declaration is an attribute only from the "naive" perspective of the pre-namespace *XML 1.0* recommendation. From the perspective of namespace-aware specifications (such as XPath, DOM, or the infoset), this is not an attribute at all but a namespace declaration: it is not an attribute node in the DOM or XPath tree, and it is not an "attribute information item" in the document's infoset.

A related fact of some importance is that the following two documents are the same from the *XML Namespaces* perspective but different from the *XML 1.0* perspective:

```
<a:doc xmlns:a="http://a.b.c.d.com"><a:p>some text</a:p></a:doc>
<b:doc xmlns:b="http://a.b.c.d.com"><b:p>some text</b:p></b:doc>
```

Because DTDs originate with *XML 1.0*, they are namespace unaware and use the *XML 1.0* perspective. This creates validation problems discussed in the next chapter.

The Syntax of Names and EBNF Productions

The same colon character (:) that separates `xmlns` from the prefix being declared is used to separate the prefix from the local name. Here again, there was a change from *XML 1.0* to *XML Namespaces*. In *XML 1.0*, a colon could appear anywhere in the name except as the first character, any number of times. In the *XML Namespaces* recommendation, a colon can appear at most once, as a separator between the namespace prefix and the local name. In the following EBNF productions, NCName is a "No-Colon Name" that does not contain a colon.

```
[6]  QName ::=  (Prefix ':')? LocalPart
[7]  Prefix ::=  NCName
[8]  LocalPart ::=  NCName
```

Most of today's parsers will reject as non-well-formed any documents with names containing more than one colon.

Scope of Declarations

Namespace declarations are inherited: the scope of a namespace declaration is the element to which it is attached, together with all its descendants, except those that declare their own namespaces. Everywhere within that scope, the names qualified by the prefix belong to the declared namespace. (In the previous XSLT example, these tag names are `stylesheet`, `output`, and `template`.) Conversely, if an element's name has a prefix but no namespace declaration, the parser will go up its line of ancestors until a declaration for that prefix is found. If no such declaration is found, the parser must return an error.

Because namespace declarations are inherited, it is possible (and recommended) to declare all namespaces on the root element, as in the example shown in Listing 2-5.

Listing 2-5. The Root Element of a Document with Three Namespaces

```
<citeDB
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xlink="http://www.w3.org/1999/xlink"
>
```

The three namespaces declared on this element are all well known. One of them is for XLink elements and attributes that will be the subject of much discussion and an extended example later in this chapter. The other two are for

Resource Description Framework (RDF) and Dublin Core Metadata that are discussed in detail in Chapter 7.

The same prefix can be mapped to different namespaces within the same document. It is therefore possible to shadow one declaration with another one in an embedded element. We have never seen a convincing case for using this functionality, but here is a contrived example (the quotes, however, are real):

```
<prfx:bk xmlns:prfx="http://chairman.mao.sayings.org.red">
  <prfx:saying>
The world is progressing, the future is bright
 and no one can change this general trend of history.
  </prfx:saying>
  <prfx:bk xmlns:prfx="http://chairman.greenspan.sayings.org.green">
    <prfx:saying>
History provides excellent lessons for banking institutions
with regard to appropriate pricing, underwriting, and diversification.
    </prfx:saying>
  </prfx:bk>
</prfx:bk>
```

The first `prfx:saying` element in this example is in the `mao.sayings.org.red` namespace, but the second such element is in the more local `greenspan.sayings.org.green` namespace. We can prove this by writing an XSLT that will extract the inherited namespace URI from both of those elements and color the text content red or green accordingly. In the process, we will see how namespace URIs are accessed in the XPath tree (not as attributes).

Namespaces in XPath and XSLT

Listing 2-6 is the stylesheet that colors Mao's sayings red and Greenspan's green. It is done in the so-called "push" style of multiple independent templates. We will discuss and use it extensively in Chapter 5. Our interest here is in the XPath functions that have to do with namespaces. (Both XPath and DOM have functions that extract the local name, the qualified name (with the prefix), and the namespace URI from a given element or attribute node in the tree.) Some of those functions are used in the highlighted part of the second template. Outside the second template, there may be details that have not yet been explained: please suspend your curiosity until Chapter 5.

Listing 2-6. Namespace Handling in XSLT and XPath

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0"
>
<xsl:output method="html"/>
<xsl:template match="/">
  <html><head><title>Two Chairmen's Wisdom</title></head><body>
    <xsl:apply-templates />
  </body></html>
</xsl:template>
<xsl:template match="*[local-name()='saying']">
  <xsl:variable name="color">
    <xsl:choose>
      <xsl:when test="contains(namespace-uri(),'red')">red</xsl:when>
      <xsl:when test="contains(namespace-uri(),'green')">green</xsl:when>
      <xsl:otherwise>blue</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <div style="{concat('color:',$color)}">
    <xsl:apply-templates/>
  </div>
</xsl:template>
</xsl:stylesheet>
```

The match attribute of that template matches all element nodes (that's what the asterisk stands for) such that their local name is `saying`. As you can see, XPath has a `local-name()` function and (a few lines farther down) a `namespace-uri()` function. Both return strings that you can work with using XPath string functions, such as `contains()`. To compute the color name, we use `xsl:choose` (the XSLT equivalent of the `switch` statement in C and derived languages).

If all namespace URIs contained the color name right after the substring `sayings.org`, we could replace the `xsl:choose` expression with code that extracts the color name from the namespace URI:

```
<xsl:variable name="color"
  select="substring-after(namespace-uri(),'.sayings.org.')"
 >
```

This would generate different colors from different data without any changes in the stylesheet.

Default Namespaces

You can save yourself a little typing by creating a default namespace that is not mapped to a prefix. The syntax is as follows.

```
<doc xmlns="http://a.b.c.d.com"><p>some text</p></doc>
```

From the namespace perspective, this element is equivalent to

```
<a:doc xmlns:a="http://a.b.c.d.com"><a:p>some text</a:p></a:doc>
```

Put differently, prefix-less element names within the scope of a default namespace declaration belong to the declared namespace. This is very different from prefix-less names that belong to no namespace at all. For any namespace-aware program, the two preceding single-line documents are completely different from

```
<doc><p>some text</p></doc>
```

Note that prefix-less attribute names remain in no namespace. The only way for an attribute to be in a namespace is by having a prefix that is mapped to that namespace. (See the next section.)

Default namespaces are useful when you have some XML data that you want to cut and paste into a new XML context. If name conflicts are a possible concern, you can create a default namespace for the data to be pasted in:

```
<insertFromData xmlns="http://www.n-topus.com/ns/temp">
  <!-- inserted prefixless data goes here -->
</insertFromData>
```

To override a default namespace declaration (that is, put an element in its scope into no namespace), you have to use a special form of the namespace declaration:

```
<doc xmlns="http://a.b.c.d.com">
<p>this element is in the "http://a.b.c.d.com" namespace</p>
<nons xmlns="">this element is in no namespace</nons>
</doc>
```

This facility may also be useful in a cut-and-paste situation.

Namespaces and Attributes

Attributes and elements are treated differently by *XML Namespaces* because attributes have a natural namespace (their owner element) and don't need extra protection. It makes no sense to put attributes in the same namespace as their owner element instead of simply leaving them in no namespace at all. Attributes are like local variables in a procedure that don't need fully qualified names to prevent name conflicts. In XSLT, the attributes of XSLT elements (such as `match` or `select`) remain local:

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

It does make sense to put attributes in a namespace of their own that is different from the containing element's namespace, especially if the attributes come from an XML language of wide application. Later in this chapter, you will see examples of XLink attributes that are used to describe links within XML data. Their (reserved) names are quite common: `type`, `title`, `href`, and so on. To protect them from conflict with unrelated attributes of the same name, they are placed into a namespace and are always used with a prefix. (It can be any prefix, but traditionally `xlink:` is used.) For examples, see the XLink section that is coming up shortly.

Attributes that are placed in a separate namespace become, in effect, "global attributes" that can be added to any element in any document to provide specific functionality. Two important groups of global attributes are those with the fixed `xml:` prefix and XLink attributes that usually are mapped to the `xlink:` prefix. We introduce them in the remainder of this section.

Attributes with the `xml:` *Prefix*

The `xml:` prefix is reserved by W3C and cannot be used by anybody else. It is declared in the `XML Namespaces` recommendation and bound to `www.w3.org/XML/1998/namespace`. Several attributes always appear with the `xml:` prefix and have a fixed meaning. We mention two: `xml:lang` and `xml:base`.

The `xml:lang` attribute can be added to any element to specify the language of that element's content. The value of the attribute is either a two-letter language code as defined in ISO 639, *Codes for the Representation of Names of Languages*, or a language identifier registered with IANA (Internet Assigned Numbers Authority), or user defined. The two-letter codes cover the most familiar languages, and some of them can be extended to indicate a regional variant: fr-ca, fr-be, and fr-ch stand for the French dialects of Canada, Belgium, and Switzerland, respectively. IANA-registered names start with "i-": i-navajo. User-defined names start with "x-": x-esperanto. For an in-depth treatment of

language identifiers in XML, see Robin Cover's Web page at `http://xml.coverpages.org/languageIdentifiers.html`.

A recent newcomer to the `xml:` family is `xml:base`. Its purpose is to define a base URI for resolving relative URIs in parts of XML documents. The value of an `xml:base` attribute must be an absolute URI; its scope is the element on which it is defined and its descendants, unless a descendant defines its own. The most common use for `xml:base` will be within `xlink:href` attributes, to resolve relative links to images, applets, form-processing programs, style sheets, and other external resources.

XLink (XML Linking Language) is a recent (June 2001) W3C recommendation, released together with *XBase*. It defines several global attributes in the XLink namespace. The namespace is usually mapped to the `xlink:` prefix, which we will use throughout the rest of the book. Although small, the XML Linking Language is quite intricate and conceptually complex because it overlays a graph structure over a collection of XML and non-XML resources. It is also a very important member of the XML family of specifications.

XLink Attributes and XLink Graphs

The purpose of XLink is to establish connections between and among resources. A resource, as usual, is anything that can be addressed with a URI. It doesn't have to be an XML resource; if it is, it does not have to be a complete document because the URI can be extended with a fragment identifier to select a document part. A very common kind of fragment identifier is an XPath expression, as you will see in a moment.

A structure that consists of nodes connected by arcs is called a *graph*. XLink is about directed graphs of resources. A graph is called *directed* if its arcs have a direction from source to target. In the case of XLink graphs, both source and target are resources. The nodes and arcs of an XLink graph can also have labels attached to them.

The most important XLink attribute is `xlink:type`. It can have several possible values, including `simple` and `extended`. An element that has an `xlink:type` attribute with one of those two values is called a `link element`. A link element can have any tag name whatsoever: it's the `xlink:type` attribute that defines it as a link element.

Link elements can be simple or extended, depending on the value of `xlink:type`. It is important to understand from the start that a link element describes an entire graph of resources, not just the arcs. The word *link* is used in its general meaning, meaning a connection or as a synonym for *arc*, but a link element is an XML element that has an `xlink:type` attribute whose value is `simple` or `extended`. From the XLink perspective, it describes a graph.

Let's take a look at a simple link element, in comparison with the HTML <a> element.

A Simple XLink "Link Element" and an HTML Hyperlink

If you think about it, the HTML <a> element describes a directed labeled arc between two resources: the source of the arc is the <a> element itself, the target of the arc is specified in the href attribute, and the label is the content of the <a> element:

```
<a href="cs303/classList.htm">cs303 class list</a>
```

The closest XLink analog would look like this:

```
<somePrefix:someElement
  xlink:type="simple"
  xlink:href="classList.xml"
  xlink:title="cs303 class list"
  xlink:actuate="onRequest">
  <!-- wait for user request to traverse the arc -->
 xlink:show="replace"
 <!-- upon traversal, replace the current document with the target -->
>
  Any text with any non-xlink markup.
</somePrefix:someElement>
```

The two links show the usual contrast between the fixed appearance and behavior of HTML and the flexibility of XML. The name of the HTML link element is fixed. Links created by that element are for human users. Their labels are automatically highlighted; even if they are not blue and underlined, which is the most common way to style them in Web pages, they have to be visible to perform their function, which is to provide a hypertext jump from the source to the target. (Such links are called *hypertext links*.) By default, the jump replaces the source document with the target in the browser window.

By contrast, an XML simple link element can have any tag name; its content doesn't have to be highlighted; it may be intended for human users or programs; and the behavior of the link is up to the application that processes it. XLink provides "behavioral attributes" that supply hints about the intended behavior, as indicated by code comments, but these are just hints and they can be ignored. Even if they are followed, the hints say nothing about blue underline, the raised cursor finger, or a single click.

For all the differences between the HTML hypertext link and the XLink simple link element, they have two important similarities:

- They describe a graph that involves just two resources and a single arc.
- The arc is outbound: its source is local (the link element itself, not specified by a URI) and its target is remote (specified by a URI).

XLink extended link elements do not have either of these restrictions. An extended link element can

- describe a complex graph containing multiple sources, targets, and arcs.
- describe arcs that are inbound (local target, remote source) or third-party (both source and target are remote).

In other words, you can create links from and between resources for which you don't have a write permission.

The natural question is: "What does one do with those links?" Hypertext links are very intuitive: you click on them, and they take you there. If there are multiple targets, though, where does the click take you? The answer, as always, is that it's up to the application to decide what to do with different kinds of links, and as of today nobody has yet come up with a killer app for multiple-target or third-party links. All we have is a flexible and powerful language to describe them.

Extended Link Element and Its XLink Graph

To specify a graph, an extended link needs to specify resources and arcs between them. These are defined by children elements of the extended link element. The names and namespaces of those elements are completely unconstrained, but they must all have an `xlink:type` attribute to indicate their role in the graph. The possible values of `xlink:type` for children elements of an extended link element are as follows:

- `locator` for "locator elements" that specify remote resources
- `resource` for "resource elements" that specify local resources
- `arc` for "arc elements" that specify arcs

The notion of a local resource is a tricky one. A local resource is an XML element that satisfies three conditions:

- It is a child of an extended link element or the link element itself.
- It has an `xlink:type` attribute whose value is `resource`.
- It does not have an `xlink:href` attribute.

A remote resource, by contrast, is a resource specified by a URI that is the value of an `xlink:href` attribute of a locator element. Put differently, for locator elements an `xlink:href` attribute is required, whereas for resource elements an `xlink:href` attribute is not allowed. The same exact XML element can be either a local or a remote resource depending on whether it is specified by a URI reference or by its position with respect to the link element.

Both locator elements and resource elements must have an `xlink:label` attribute that contains an identifying label. The label has to be an NCName; that is, it cannot have a prefix and a colon. Arc elements (the children of an extended link element that have `xlink:type="arc"`) refer to their source and target resources using those labels. In summary, the XML structure of an extended link element looks as shown below. In the outline, `elt` stands for an arbitrary tag name, `uri` for an arbitrary URI, and `lbl` for an arbitrary label. There have to be two or more resources and at least one arc. The order of children is not constrained by XLink itself but can be constrained by a DTD or some other type of schema or grammar.

```
<elt xlink:type="extended" . . . ><!-- possibly other attributes -->
  <!-- external resources / locator elements -->
  <elt xlink:type="locator"
      xlink:href="uri"
      xlink:label="lbl"/>
  <!-- local resources / resource elements -->
  <elt xlink:type="resource"
      xlink:label="lbl"/>
  <!-- arcs / arc element -->
  <elt xlink:type="arc" xlink:from="lbl" xlink:to="lbl" />
</elt>
```

This outline contains only structural markup that defines the graph. XLink also has behavioral and semantic markup.

Behavioral and Semantic Markup

You have already seen behavioral attributes, `xlink:show` and `xlink:actuate`. Their values are largely self-explanatory. The specification spells out in some detail

what the conformant applications should do in the presence of behavioral markup. There are no required actions, only recommended ones:

```
show: "new", "replace", "embed", "other", and "none"
actuate: "onLoad", "onRequest", "other", and "none"
```

Semantic attributes are `xlink:title`, `xlink:role`, and `xlink:arcrole`. The `title` attribute can appear on any element to provide a brief description. The `role` attribute can appear on extended, simple, locator, and resource elements. The `arcrole` attribute can appear on simple and arc elements. Both `xlink:role` and `xlink:arcrole` must be absolute URI, perhaps with a fragment identifier attached. Their intended use is rather vaguely defined, but see the following RDDL section for an example.

In addition to the `xlink:title` attribute, extended link-, locator-, and arc-elements can have any number of children elements of type `title`, that is, elements with `xlink:type="title"`. Their purpose is to provide a more structured and extensive annotation or a series of annotations, perhaps in different languages. (The value of an attribute can only be CDATA.)

Summary of XLink Attributes

Table 2-2, quoted from the XLink specification, shows all XLink attributes and how they coexist with different values of `xlink:type`, listed as column headers. Now that you have seen them all, this table may actually be useful. (*R* stands for *required*, *O* stands for *optional*, and the *N/A* stands for *not applicable*.)

Table 2-2. Summary of XLink Attributes

ATTRIBUTE	SIMPLE	EXTENDED	LOCATOR	ARC	RESOURCE	TITLE
type	R	R	R	R	R	R
href	O	N/A	R	N/A	N/A	N/A
role	O	O	O	N/A	O	N/A
arcrole	O	N/A	N/A	O	N/A	N/A
title	O	O	O	O	O	N/A
show	O	N/A	N/A	O	N/A	N/A
actuate	O	N/A	N/A	O	N/A	N/A
label	N/A	N/A	O	N/A	O	N/A
from	N/A	N/A	N/A	O	N/A	N/A
to	N/A	N/A	N/A	O	N/A	N/A

An XLink Example

The following document shows an extended link element with third-party arcs. For our examples, we chose a Bible commentary. The Bible itself, as you know, contains many more-or-less obvious cross-references within itself, from later to earlier books. It also contains elaborate rhetorical and narrative patterns that take time and study to discover, and Bible commentaries point out those cross-references and patterns. These commentaries also contain cross-references to themselves and other commentaries, making it all a very tangled graph indeed.

We will do a simple example of a narrative pattern within the Bible itself, primarily because we couldn't find any XML-marked Bible commentaries. As XML source, we use ot.xml, the King James Version marked up by Jon Bosak and distributed as part of his Religious Works package (www.ibiblio.org/bosak/). The package contains tstmt.dtd, which defines the markup structure. Both the XML source and the DTD can be downloaded from this book's Web site, but the following XPath expressions (within XLinks) should be self-explanatory without the DTD: the root element is tstmt, which contains bookcoll elements (book collections), which contain book elements, which contain chapter elements, which contain v elements (verses). To select verses 5 through 11, we say [position()>4 and position()<12]. In an XML file, we encode ">" and "<" as > and <.

We assume two namespaces, one for XLink mapped to xlink: and the other for the text of the commentary, mapped to c:. We do a single extended link element, c:comm, but you should think of it as a child of the root element, c:commentary, that contains an arbitrary number of c:comm extended link elements. Such collections of third-party links are called *linkbases*, and we can say that we show (and process) a minimal linkbase.

An Extended Link

Our example (see Listing 2-7) comes from the story of Joseph (Genesis 37-49) that contains three dream sequences, each consisting of two dreams. We will describe this by an extended link element that has three locator elements (one for each dream sequence) and six arcs, connecting each dream sequence to the other two.

Because the code is quite repetitious, we show only one locator element and one arc element. All namespaces are declared on the root element. The value of the xlink:href attribute is a very long string that, on the printed page, has to be broken into three lines.

Listing 2-7. An Extended Link Example from dreams.xml

```
<c:comm xlink:type="extended" xlink:title="Dreams in the story of Joseph">
  <c:txt>There are three dream sequences in the story of Joseph,
                Joseph's dreams,  Prisoners' dreams and Pharaoh's dreams.</c:txt>
<c:node type="narrativePattern"
  xlink:type="locator"
  xlink:title="Joseph's dreams"
  xlink:label="Jdreams"
  xlink:href=
    "http://localhost:8080/xmlp/dat/jb/ot.xml
      #xpointer(/tstmt/bookcoll/book[bktshort='Genesis']
                 /chapter[37]/v[position()&gt;4 and position()&lt;12])">
Joseph tells his brothers about his dreams. The dreams predict that the brothers
will bow to Joseph and become subservient to him. The brothers are not happy.
</c:node>
<!-- two more nodes like this -->
<c:crossRef xlink:type="arc" from="Jdreams" to="Pdreams"  </crossref>
<!-- five more arcs like this -->
</c:comm>
```

The new material in this example is the fragment identifier that follows the URI in the multiline `xlink:href` attribute. It is a single string that is again broken into three lines on the printed page:

```
"http://localhost:8080/xmlp/dat/jb/ot.xml
  #xpointer(/tstmt/bookcoll/book[bktshort='Genesis']
    /chapter[37]/v[position()>4 and position()<12])"
```

The fragment identifier consists of two parts: a URI and an XPointer expression. (XPointers are defined in `www.w3c.org/tr/xptr`.) The XPointer expression is the function `xpointer()` whose argument is an XPath expression. Together, the URI and the XPath expression uniquely identify a node set on the Internet. In our example, the expression says

> *at the top-level, pick the "tstmt" element; within that look for a "bookcoll" which contains a "book" whose "bktshort" (book-title-short) subelement is "Genesis"; from this book take the 37th "chapter" element; within this chapter take every verse whose position is > 4 and < 12.*

Note that predicates that constrain the set of nodes selected by an XPath or XPointer expression appear in square brackets after the tag name, as in

`[bktshort='Genesis']` or `v[position()>4 and position()<12]`. Because this is an XML document, angle brackets are encoded as `>` and `<`.

Note on XPointers

XPointer expressions are mostly XPath expressions, with two additions:

- Expressions that refer to a specific point between two characters in the text content of the document, and
- Expressions that refer to character ranges within the text content of the document.

All of the XPointers in this chapter are also XPaths.

It is anticipated that XPointers will typically be used in XLink elements to indicate link endpoints. When used that way, the XPointer expression is given as an argument to the `xpointer()` function, as in our example.

An XLink Application

Our first XLink application does not do much; it does not even output any blue underlined links to click on. However, it does process an extended link in a general way, extracting all the information it contains, including XML data referenced by XPointers. It is a Web application that uses Java Server Pages (JSPs) and XSLTs and runs in Tomcat. The XSLTs it uses introduce some useful general-purpose techniques.

The application assumes that there is a source of XML data that is not subject to change (the King James Bible). A separate “linkbase” file contains extended links that are cross-references within the source. The entry page to the application is the familiar xx.jsp that expects, as you recall, two arguments: an XML file and an XSLT to apply to it. In this application, the XML file is our linkbase and the XSLT is a data-specific program, dreams.xsl. It incorporates, by inclusion, a data independent, general purpose XLink application called linkTransform.xsl. This is a fairly complex program that is partially discussed in the next section; a complete explanation will have to wait until Chapter 5. We use it in this chapter to provide an interesting example of XLink processing. You can experiment with dreams.xsl and the linkbase even if you skip the next section altogether.

Schematically, the application consist of components shown in Figure 2-6. The components above the broken line are completely explained in this chapter and can be experimented with. The components underneath the broken line are briefly explained in the next section and completely explained in Chapter 5.

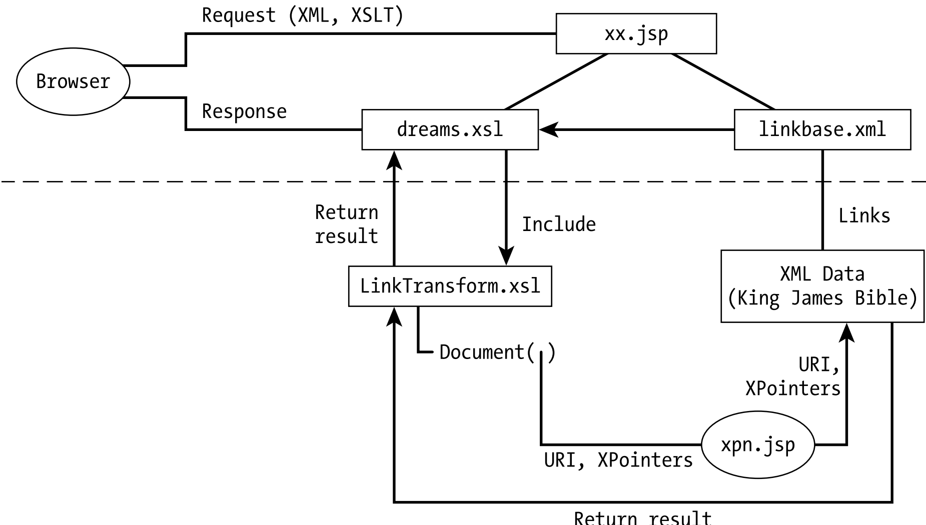

Figure 2-6. An XLink application with extended links

The data-specific dreams.xsl is quite readable:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 version="1.0">
<xsl:output method="html"/>
<xsl:include href="linkTransform.xsl"/><!-- include another file -->
<xsl:template match="v">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Remember that our task is to understand the syntax and semantics of XLink and the general outlines of the XSLT that processes it. The complete details of XSLT are quite involved, but the general outlines are clear. In dreams.xsl, just note that the code expects to receive some material from linkTransform.xsl that contains v elements in it. (This is the import of <xsl:template match=”v”>) All it does is outputs those elements as XHTML `<p>` elements.

linkTransform.xsl

All the real work is done in linkTransform.xsl, which knows nothing about the data source. In outline, this is what’s happening. The linkTransform.xsl file extracts information from the linkbase, including the `xlink:href` attributes. These attributes, as we just discussed, contain the data source URI and an

XPointer. After some fairly elaborate footwork (which is discussed in detail in the next section), the XSLT sends both the URI and the XPointer to another JSP application. That other JSP, xpn.jsp, returns the data referenced by the XPointers. The data consists of v elements that contain Biblical verses. Eventually, that data ends up in dreams.xsl that converts verses to paragraphs.

For example, with Tomcat running and all the files in the right places, this URL (shown broken over two lines)

```
http://localhost:8080/xmlp/xx.jsp?
     xmlUri=helloXLink/dreams.xml&xslUri=helloXLink//dreams.xsl
```

results in the screenshot shown in Figure 2-7.

Figure 2-7. XLink processed with XSLT (Joseph's Dreams)

The Code of linkTransform.xsl

This XSLT can be classified as advanced: it goes beyond what we have so far covered and can be skipped on first reading. We don't discuss all of it but we do address two details that concern calling xpn.jsp and sending parameters to it. First, we introduce the XPath `document()` function. Its main use is to include another XML document for processing. So, for instance, if you have an XML document additionalData.xml and you want to store its XPath tree in a variable in your XSLT program, you would say

```
<xsl:variable name="moreData" select="document('additionalData.xml')" />
```

The `document()` function takes any URI as argument, including URIs that connect to JSPs and are followed by a query string. In other words, the `document()` function is perfectly happy to connect to a JSP application and send it some arguments. For instance, at some point in linkTransform.xsl, we say

```
<table border="1"><tr><td>
 <xsl:apply-templates select="document(concat($uri,$qstring))/*" />
  </td></tr></table>
```

This results in the following sequence of events.

- The values of two variables, `uri` and `qstring`, are concatenated.
- The result is an argument to a document() call that in turn is a call on xpn.jsp.
- The call returns some material—a single element called `<nodeset>`—from the XML data source using an XPath expression. (Both are specified in the `qstring`.)

All children elements of the extracted element are processed by whatever templates will match them in the containing stylesheet that knows about the structure of the data. In our case, it's dreams.xsl that outputs the Bible text in the bordered box in the screenshot.

The two concatenated variables are declared as follows.

```
<xsl:variable name="uri" select="'http://localhost:8080/xmlp/xpn.jsp?'"/>
<xsl:variable name="qstring" select="concat('x=',$x-enc,'&amp;p=',$p-enc)"/>
```

The first declaration is straightforward, but note the single quotes within double quotes: the double quotes enclose the XML attribute, and the single quotes enclose the constant string that becomes the value of the variable. Without single quotes, the XSLT processor would try to evaluate the string as an XPath expression.

The `qstring` declaration concatenates some constant strings and variable values. Its purpose is to produce the following query string, shown broken over two lines.

```
x=ot.xml&p=/tstmt/bookcoll/book[bktshort='Genesis']/chapter[37]
            /v[position()>4 and position()<12]
```

Before this string can be sent to the server, it has to be appropriately encoded. What does *appropriately encoded* mean? As you know, one can add

parameters to a URI, separated by a question mark, as you saw in Chapter 1 with xx.jsp (the URI is divided into two lines):

```
http://localhost:8080/tryXSL/xx.jsp?
    xmlUri=helloXSL/hello.xml&xslUri=helloXSL/hello.xsl
```

The technical name for the part of the URI that follows the question mark is *query string*, because it is intended for sending queries to Web applications. The problem with the query string is that it can contain only characters that are allowed in a URI. The forbidden characters (including, for instance, square brackets) have to be URL-encoded. The URL encoding of a character consists of a "%" followed by two hexadecimal digits showing the UTF-8 code for the character. Because these are single-byte characters, UTF-8 codes are the same as ASCII codes: space comes out as `%20`, left bracket as `%5B`, and right bracket as `%5D`.

XSLT/XPath Extension Functions

Instead of doing URL encoding by hand, we call a function to do that. XPath itself does not have such a function but most XSLT processors have a facility for adding your own extension functions. This facility will be standardized in the next release of XSLT, but even now it works pretty much the same way in different implementations. Follow these two steps if you want to call a static method of a Java class:

1. Declare a namespace which, for Xalan, is `xmlns:java="http://xml.apache.org/xslt/java"`. Note that this is a specialized use of namespaces, completely unrelated to their use in general-purpose XML documents, as opposed to XSLT programs.

2. Within XPath, put the namespace prefix before your function call:

   ```
   java:java.net.URLEncoder.encode().
   ```

Because we use a built-in Java function, we don't have to write any code. The function is a public static method of the `java.net.URLEncoder` class.

This is how we extract the XPointers from dreams.xml and encode them:

```
<xsl:variable name="h" select="@xlink:href"/>
<xsl:variable name="x" select="substring-before($h,'#xpointer')"/>
<xsl:variable name="p" select="substring-after($h,'#xpointer')"/>
<xsl:variable name="x-enc" select="java:java.net.URLEncoder.encode($x)"/>
<xsl:variable name="p-enc" select="java:java.net.URLEncoder.encode($p)"/>
```

The preceding section shows how `x-enc` and `p-enc` are used to construct `qstring`. The entire linkTransform.xsl file is shown in Listing 2-8, with the part we have discussed highlighted. We've also highlighted the beginning of every template, of which there are several. The stylesheet uses the `xsl:apply-templates` construct extensively. We will discuss the construct and the programming style based on it in Chapter 5. In the meantime, you can experiment by simply changing the XML file that contains your XLinks.

Listing 2-8. XSLT Stylesheet to Process XLinks

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:c="http://n-topus.com/ns/jdreams"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns:java="http://xml.apache.org/xslt/java"
                version="1.0">
<xsl:output method="html"/>
<xsl:template match="/"><-- match root, apply templates to its children -->
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="c:comm">
  <xsl:variable name="title" select="@xlink:title"/>
  <html> <head> <title>  <xsl:value-of select="$title"/> </title>
  </head>
    <body >
      <h1><xsl:value-of select="$title"/></h1>
        <xsl:apply-templates/>
      </body>
    </html>
 </xsl:template>
<xsl:template match="c:txt">
  <p><strong><xsl:apply-templates/></strong></p>
</xsl:template>
<xsl:template match="c:node">
  <h2><xsl:value-of select="@xlink:title"/></h2>
  <p><span style="color:green">
       <xsl:apply-templates/>
    </span>
  <xsl:variable name="h" select="@xlink:href"/>
  <xsl:variable name="x" select="substring-before($h,'#xpointer')"/>
  <xsl:variable name="p" select="substring-after($h,'#xpointer')"/>
  <xsl:variable name="x-enc" select="java:java.net.URLEncoder.encode($x)"/>
  <xsl:variable name="p-enc" select="java:java.net.URLEncoder.encode($p)"/>
  <xsl:variable name="qstring" select="concat('x=',$x-enc,'&amp;p=',$p-enc)"/>
  <xsl:variable name="uri" select="'http://localhost:8080/xmlp/xpn.jsp?'"/>
```

```
  <table border="1"><tr><td>
   <xsl:apply-templates select="document(concat($uri,$qstring))/*" />
    </td></tr></table></p>
</xsl:template>
</xsl:stylesheet>
```

The last three lines before the closing tag basically say "get the XPointer-referenced stuff out of the data source, apply whatever templates apply to it in the containing XSLT, and put the result in a box on screen." In our minimal stylesheet, there is only one simple template, but the document-specific processing can be as complex as it needs be. This is the signature feature of the `apply-templates` processing model.

The JSP Page

The remaining software module is the JSP that uses XPath to extract a node-set. Its operation consists of these steps:

1. Parse the data source.
2. Obtain a DOM object.
3. Call a `selectNodeList()` method to extract the list of nodes that satisfies the XPath/XPointer condition. The method takes two arguments: the root of the subtree to apply XPath to and the XPath expression to use.

Because the data source remains unchanged, an obvious optimization is to parse it once and cache the resulting DOM tree in an object that persists from one application call to another. Listing 2-9, xpn.jsp, illustrates this idea. In JSP, a simple way to obtain a persistent object is to make it "application scope." Similar functionality is available in ASP.

Listing 2-9. JSP Page with XPath

```
%@ page errorPage="error.jsp"
import="org.apache.xalan.xslt.*,org.apache.xerces.parsers.*,
org.w3c.dom.*,org.apache.xml.serialize.*"
%><jsp:useBean id="cache" class="java.util.Hashtable" scope="application"/><%
  String xPath=request.getParameter("p");
  String xml=request.getParameter("x");
  if(0>xml.indexOf(":"))  xml="file:///"+application.getRealPath(xml);
  Document doc=(Document) cache.get(xml);
  if(doc==null){ // has not been parsed yet
```

```
    DOMParser parser=new DOMParser();
    parser.parse(xml);
    doc=parser.getDocument();
    cache.put(xml,doc);
  }
  NodeList ndList=org.apache.xpath.XPathAPI.
           selectNodeList(doc.getDocumentElement(),xPath);
  if(ndList==null){ // format and display an error message, see the code file
  } else {
  OutputFormat outputFormat=new OutputFormat();
  outputFormat.setOmitXMLDeclaration(true);
  // this XML data is a fragment, no declaration,
  // can be spliced into another document
  XMLSerializer ser= new XMLSerializer(out,outputFormat);
%><nodeList><%
  for(int i=0;i<ndList.getLength();i++)
     ser.asDOMSerializer().serialize((Element)ndList.item(i));
%></nodeList><%
}
%>
```

A similar Web application can be made using ASP: both JScript and VBScript have all the relevant functionality, including querying DOM trees with XPath expressions. This feature will become standard in *DOM Level 3*, currently under development.

Namespace Controversies and RDDL

Unlike other early XML recommendations (*XML 1.0*, *DOM Level 1*, *XSLT*, and *XPath*), *XML Namespaces* provoked a lot of discussion and argument. Until *XML Schema* came along, it was easily the most controversial of XML specifications. It also provoked a good deal of confusion because XML namespaces were very different from the familiar programming language namespaces, sometimes in counterintuitive ways.

Namespace Explanations

Fortunately, in response to confusions and controversy, many illustrious people wrote perceptively about namespaces, including Tim Bray, David Megginson, and James Clark. All their contributions are online, together with an excellent

FAQ by Ronald Bourret, who also wrote a separate piece exploding namespace myths. Here is a list of resources from which our own summary is synthesized.

- *XML Namespaces* by James Clark (`www.jclark.com/xml/xmlns.htm`)
- *XML Namespaces by Example* by Tim Bray (`www.xml.com/pub/1999/01/namespaces.html`)
- *19 Short Questions about Namespaces (with Answers)* by David Megginson (`www.megginson.com/docs/namespaces/namespace-questions.html`)
- *Namespace Myths Exploded* by Ronald Bourret (`www.xml.com/pub/2000/03/08/namespaces/index.html`)
- *Namespaces FAQ* by Ronald Bourret (`www.rpbourret.com/xml/NamespacesFAQ.htm`)

A Brief List of Confusions and Controversies

Of the many items that have come up in multiple discussions, we have chosen three that we think are the most important.

- Unlike in programming languages, the names of XML namespaces are different from the prefixes that qualify local names. Those prefixes (with the exception of `xml:`) are arbitrary. The same prefix can map to different namespaces within the same document, and the same namespace can map to different prefixes.
- *XML 1.0* pre-dates *XML Namespaces*. It treats namespace declarations as any other attribute; it treats qualified names as monolithic strings with no internal structure, and it knows nothing about the relationship between namespace prefixes and namespace URIs. One consequence of this is that DTDs and later specifications such as *DOM* or *XPath* have different ideas about element names and document identity.
- Programming language namespaces are real and contain information about their names. A Java or C++ class contains declarations and definitions of the names that its own name qualifies. If your class is called Address, and you have a variable named postalIndex, then the qualified name of this variable is Address.postalIndex; but, also, the definition of the Address class declares that variable and specifies some of its properties

> (such as data type and initial value). By contrast, XML namespace URI, contrary to its name (Resource Identifier) does not identify any resource. There is no commitment whatsoever that de-referencing that URI will get you to any place reasonable, or any place at all. According to the namespace recommendation, the namespace URI has no intended meaning: it's just a unique string of characters that protects against name conflicts.

This last feature of XML namespaces has been especially difficult to accept. A number of people argued that a namespace URI should point to a DTD, or an XML schema, or some such resource that would provide information about the syntax of the names that belong to that namespace, and perhaps also about their intended meaning. However, it proved impossible to build a consensus on what such an authoritative resource would be, and many argued that XML's unique decentralized strength is in having its intended interpretation left unconstrained by anything authoritative.

RDDL to the Rescue

In the end of 2000, Jonathan Borden and Tim Bray developed a compromise proposal that seems to be gaining acceptance. Instead of a specific resource, they proposed that the namespace URI should point to a resource-description document that would describe standard resources (such as stylesheets, schemas, and so on) in a standard format. They called the format *RDDL* (*Resource Directory Description Language*) (For more information, visit `www.rddl.org`.)

By design, an RDDL document is human-readable and machine-processable. For humans, RDDL looks just like XHMTL. To give machines something to do, RDDL has a single additional element called `resource`. The resource element of RDDL is also a simple link element in the XLink sense: it has a required `xlink:type` attribute whose value, in the current version of the specification, can only be `simple`. (However, when extended link processors are widely available, it will probably be common to find an RDDL resource element that is an extended link element and has XLink resource elements as its children.)

Here is an RDDL example, a document describing resources for the pdata.xml example of Listing 1-7. We repeat here the beginning of that example, with a default namespace declaration added.

```
<?xml version="1.0"?>
<!-- personal data for people and other kinds of personalities -->
<pdata xmlns="http://csproj.colgate.edu/xmlp/ns/pdata/">
...
</pdata>
```

The namespace URI points to an existing directory, `http://csproj.colgate.edu/xmlp/ns/pdata/`, within which the RDDL file, index.html, is the default. The RDDL file, shown in Listing 2-10, follows this outline:

- root element with namespace declarations
- introductory prose
- validating resources (DTD, XML Schema, RELAX NG)
- display resources (CSS)

Listing 2-10. An RDDL Example

```
<!DOCTYPE html PUBLIC "-//XML-DEV//DTD XHTML RDDL 1.0//EN"
                      "http://www.rddl.org/rddl-xhtml.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:rddl="http://www.rddl.org/">
<head><title>PData Resources</title></head><body>
<h1>PData Resources</h1>
<p>
  This is a resource file for the <tt>pdata</tt> example of chapters 1 and 2 of
  <cite>XML for Programmers</cite> by Alexander D. Nakhimovsky and Tom Myers.
  The standard namespace for this example is
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/">
    http://csproj.colgate.edu/xmlp/ns/pdata/</a>
  and that directory's default,
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/index.html">
    http://csproj.colgate.edu/xmlp/ns/pdata/index.html</a>,
should  be the URL of the current version of this file.
</p>
<h2>Validation</h2>
<p>You can validate a <tt>pdata</tt> document with a dtd,
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd">
    http://csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd</a>.
</p>
<rddl:resource
  xlink:title="DTD for pdata validation"
  xlink:href="http://www.csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd"
  xlink:role="http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml-
dtd"
  xlink:arcrole="http://www.rddl.org/purposes#validation"
>
```

```
  <p>A sample DOCTYPE would be</p>
<pre><tt>&lt;!DOCTYPE pdata  PUBLIC "-//Nakhimovsky-Myers//DTD pdata 0.05//EN"
           "http://www.csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd"&gt;
</tt></pre>
</rddl:resource>
<p>You can also validate a <tt>pdata</tt> document with a RELAX NG grammar.</p>
<rddl:resource
  xlink:title="RELAX NG grammar for pdata validation"
  xlink:href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng"
  xlink:role="http://www.rddl.org/#resource"
  xlink:arcrole="http://www.rddl.org/purposes#validation"
>
<p>Such a grammar is available at
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng">
    http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng</a>
 </p>
</rddl:resource>
<h2>Display</h2>
<rddl:resource
  xlink:title="CSS Style Sheet for pdata display"
  xlink:href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css"
  xlink:role=
 "http://www.isi.edu/in-notes/iana/assignments/media-types/text/css"
>
 <p>
   There is a simple CSS1 style sheet for pdata documents at
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css">
      http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css</a>
 </p>
</rddl:resource>
</body>
</html>
```

As you can see, each resource element is a simple link element with `xlink:role` and `xlink:arcrole` attributes on it (in addition to the required `xlink:href`). Guidelines on how to use these two attributes (to specify the nature and purpose of the resource, respectively) can be found at `www.rddl.org`. Some of the common natures and purposes are also listed there. The intent is to provide enough information to make RDDL documents processable by machines.

Some of the required processing is fairly straightforward. Consider DTD validation. The “RDDL standard” values for the nature and purpose attributes are as follows.

```
xlink:role=
    " http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml-dtd"
xlink:arcrole="http://www.rddl.org/purposes#validation"
```

Given a namespace URI, we extract the document at the end of it (with the `document()` function in XSLT) and select an RDDL resource element with the "validation" `arcrole`. The XPath expression would be

```
//rddl:resource[@xlink:arcrole='http://www.rddl.org/purposes#validation']
```

If we can confirm that this element has the DTD nature as well, then `xlink:href` must be a link to a DTD that can be used for validation in ordinary ways, as presented in the next chapter.

Similarly, we can look for other standard resources with other standard roles. It seems plausible that systems of such roles will be developed: with XML used to describe commercial objects, we can expect a "purchase" role for a resource that helps you link to software for buying one of whatever it is, and a "complaints-department" role that helps you link to software to say that what you bought wasn't what you thought you were buying.

Conclusion

We've covered a lot of ground in this chapter. The most important notion is a well-formed document, and the second most important notion is a namespace. We have seen several XML languages, including XHTML, XLink, and RDDL. What we have not done in this chapter is pose a question such as "How do we check that a particular XHTML document is not only well formed but also contains only the markup that is expected in XHTML?" This is the question of validity or conformance to a specific grammar, and we take it up in the next chapter.

CHAPTER 3

DTDs and Validation

THIS CHAPTER WILL DISCUSS DTDs in considerable detail. We start with three frameworks for testing before proceeding to examples and discussion. Most of the chapter covers "advanced" topics having to do with the use of entities, namespace handling, and modularization. Throughout the chapter, we compare specific features of DTDs with more-recent approaches to validation (RELAX NG and XML Schema). By the end of the chapter, having covered the foundations (*XML 1.0* and *XML Namespaces*), we review the infoset, which is currently a W3C candidate recommendation.

In outline, the chapter proceeds as follows:

- DTDs and validation
- What else do DTDs do? Macros, inclusions, and XHTML modularization
- What's wrong with DTDs?
- summary of *XML 1.0*, *XML Namespaces*, and *Infoset*

DTDs and Validation

XML inherited DTDs from SGML and simplified their notation but preserved most of their functions. The main function of a DTD (and the task from which it derives its name) is to define a document type together with its elements and attributes. This part of DTD is a grammar for the XML language it defines. As an example, consider again the DTD of Chapter 1's Listing 1-5.

```
<!DOCTYPE exchange [
<!ELEMENT exchange (q, a)>
<!ELEMENT q (#PCDATA)>
<!ELEMENT a (#PCDATA)>
<!ATTLIST exchange tone (friendly|polite|cold|rude) "friendly">
]>
```

Also in Chapter 1, we showed two documents (Listing 1-3 and 1-4), both well formed, but with one conforming to this DTD and the other violating its rules. In Listing 1-4, the order of `<q>` and `<a>` is reversed, and the `tone` attribute has a value that is not listed in the definition. Remembering the terminology, a well-formed document that also checks out against its DTD is called *valid* with respect to that DTD. A parser that understands the DTD language is called a *validating parser.* Because you frequently don't want to validate (because the DTD is not available or to avoid performance penalty), validating parsers typically have a settable Boolean flag that turns validation on and off. Unfortunately, XML-aware browsers, even if they carry a validating parser, do not provide such a flag, and simply do not support validation as part of displaying the file. If you want to validate and display, validation has to be done from within a program.

Running a Validating Parser

We provide three applications that validate and display either the document or an error message:

- Windows/IE-specific client-side application using JScript
- Web application using JSP and Tomcat
- Web application using ASP and PWS/IIS

The client-side application is the simplest. To run it, open TestValidityJS.htm as a local file, enter an XML filename, and click on the Test button. Either the file source or an error message will be displayed in the text area of TestValidityJS.htm (as seen in Figure 3-1).

> **NOTE** *If you are using IE5.5 with MSXML 3.0, you have to enable the security option "Initialize and script ActiveX controls not marked as safe" on the local intranet domain.*

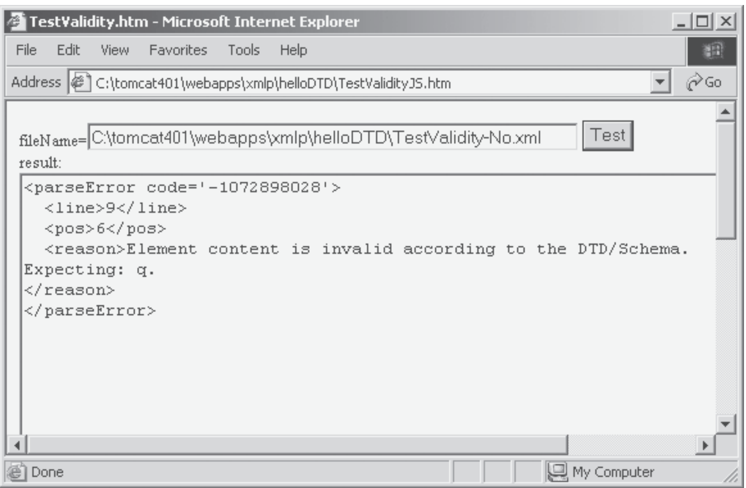

Figure 3-1. Error message from the client-side validator

To run the Web applications (ASP or JSP), you enter this URL, with a filename in the end. The first line (shown here divided into two lines for display purposes) calls IIS on port 80, and the second line (also shown divided) calls Tomcat on port 8080.

```
http:// localhost/xmlp/helloDTD/Validate.asp?
  xml=TestValidity-Yes.xml
http:// localhost:8080/xmlp/helloDTD/validom.jsp?
  xml=helloDTD/TestValidity-Yes.xml
```

Although the screens look nearly identical for either server and the file layout is identical for the two servers, the invocations are not quite identical. The ASP is in the same directory as the XML file, in this case inetpub\wwwroot\xmlp\helloDTD, and the JSP is in the same directory as a copy of the same XML file, TOMCAT_HOME\webapps\xmlp\helloDTD, so you would expect the file references to be identical, but ASP relative file reference starts at the ASP's location whereas JSP relative file reference starts at the JSP's "webapp," in this case xmlp itself. It shouldn't be a major issue, but it's worth remembering that these are almost—but not quite—equivalent setups. If the order of elements is wrong, you get the error message shown in Figure 3-2.

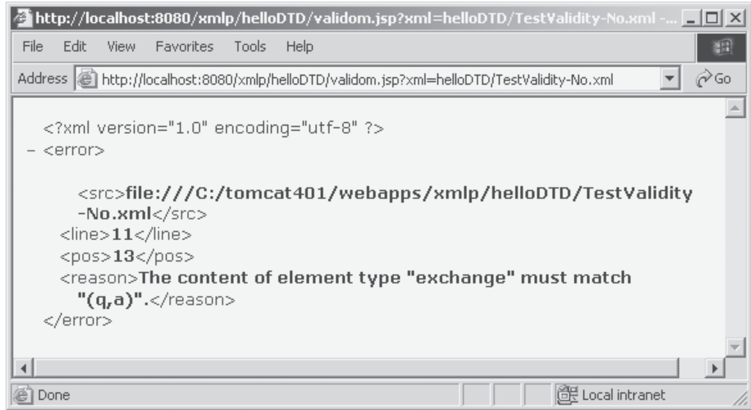

Figure 3-2. Error message from Web application validator

Here is the code for all three programs, to show similarities and differences. The JScript code is the most compact and goes first.

validateJs.js

This program (shown in Listing 3-1) consists of two pieces, an HTML file with a form and a JScript file that performs parsing and validating. The HTML file includes the JScript file by reference. The form within the file has an input box for the name of the file to be validated, and a text area to show that file if it is valid or an error message if it is invalid.

Listing 3-1. HTML File for Client-Side Validation

```
<html><head>
<title>TestValidityJS.htm</title>
<script language="JScript" src="validateJs.js"></script>
</head><body>
<form>
  fileName=<input type="text" name="fileName" value="TestValidity-Yes.xml">
  result=<textarea name="result" rows="10" cols="80">
  </textarea>
  <input type="button" value="Test"
    onclick="with(this.form) result.value=returnValidXML(fileName.value)">
</form>
</body></html>
```

The JScript code returns either an error message or the XML source to the text area. It seems impossible to display the returned XML document with a stylesheet (default or otherwise). If it were an HTML document, we could display it in a separate frame or insert it into the current file using `innerHTML="some string"`. Neither of these options is available for XML documents.

The code consists of two functions: the main function does the parsing and validation, and a supporting function constructs an error message. The main function proceeds as follows: obtain a parser object, set its validation flag to True, parse, and return either the serialized document or an error message.

```
function returnValidXML(fileName){
  var xmlObj = new ActiveXObject("MSXML2.DOMDocument");
  xmlObj.validateOnParse = true;
  xmlObj.load(fileName);  // parse!
  var res=getParseErrorXML(xmlObj);
  if(res=="") return xmlObj.xml.toString(); // serialize!
  else return res;
}
```

The error-message function extracts various pieces of information from the error object and concatenates them, together with markup, into a message to be displayed.

```
function getParseErrorXML(xmlObj) { // construct error message
  with(xmlObj.parseError){
    if(errorCode == 0)return "";
    var res=
       "<parseError code='"+errorCode+"'>\n"+
         "  <line>"+line+"</line>\n"+
         "  <pos>"+linepos+"</pos>\n"+
         "  <reason>"+reason+"</reason>\n"+
        "</parseError>";
    return res;
  }
}
```

Validate.asp

An ASP-based Web application that validates on the server does recognizably the same things, except it gets the XML filename from Request and writes the result to Response. A separate function constructs an error message, as shown in Listing 3-2.

Listing 3-2. ASP File for Validation

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Function errorXML(xmlDoc)
  If xmlObj.parseError.errorCode = 0 Then
    errorXML = ""
 Else
    errorXML =                                                      _
        "<error type='"& xmlObj.parseError.errorCode  & "'>" _
      & "  <line>" & xmlObj.parseError.line &  "</line>"     _
      & "  <pos>" & xmlObj.parseError.linepos & "</pos>"     _
      & "  <reason>" & xmlObj.parseError.reason & "</reason>"_
      & "</error>"
  End If
End Function
```

With an error function in place, we get the filename from Request, obtain an instance of the parser, set validation to True, and parse. If there is an error message, we parse the XML string produced by the `errorXML()` function. Either the parsed error message or the parsed file gets written to Response.

```
  Response.contentType="text/xml"
  Dim xmlFile,xmlObj
  xmlFile = Request.QueryString("xml")
  xmlFile = server.MapPath(xmlFile)
  Set xmlObj = CreateObject("MSXML2.DOMDocument")
  xmlObj.validateOnParse = true
  xmlObj.load xmlFile ' parse!
  If(xmlObj.parseError.errorCode <> 0) Then
    xmlObj.loadXML(errorXML(xmlObj))
  End If
  Response.write(xmlObj.xml)
 %>
```

validom.jsp

This JSP page (shown in Listing 3-3) is almost entirely Java code that is very similar to ASP in outline. After importing a few classes, it sets up an error handler for the parser it uses. (The parser has a default error handler but it just silently suppresses all error messages.) Note that, in the case of Apache Xerces and many other parser packages, two parsers are involved: a DOM parser that returns a DOM document object and a SAX parser that does the actual work of parsing

the text and identifying elements, attributes, the text content, and so on. The output of the SAX parser is used to construct a DOM object. This will be further explained in the chapters on XML parsing.

Listing 3-3. JSP File for Validation

```
<%@ page  errorPage="error.jsp" contentType="text/xml"
   import="org.apache.xerces.parsers.*, org.apache.xerces.dom.*,org.xml.sax.*,
           org.apache.xml.serialize.XMLSerializer"
%><%
  ErrorHandler errorHandler=new org.xml.sax.ErrorHandler() { // echo errors
    public void warning(SAXParseException ex)throws SAXException{throw ex;}
    public void error(SAXParseException ex)throws SAXException{throw ex;}
    public void fatalError(SAXParseException ex)throws SAXException{throw ex;}
    };
  String outputString=""; // for error message
```

With the error handler in place, we obtain an instance of the parser, set its error handler to the one we've just constructed, and set its validation feature to True. We obtain the filename from the request object and construct a complete local file URI out of it. At this point, we are ready to say `parser.parse(xml)`. If the parse is successful, we get a DOM document object and serialize it using an Apache-specific XMLSerializer class. (This is another facility that will probably be integrated into the next release of DOM.) If the parse is not successful, we construct an error message and display it. The highlighted comments in the code indicate the boundaries of the two branches of execution.

```
  DOMParser parser=new DOMParser();
  parser.setErrorHandler(errorHandler);
  parser.setFeature( "http://xml.org/sax/features/validation", true);
  String xml="file:///"+application.getRealPath(request.getParameter("xml"));
   try{ // parse and serialize
     parser.parse(xml);
     org.w3c.dom.Document doc=parser.getDocument();
     XMLSerializer ser=new XMLSerializer(out,null);
     ser.serialize(doc);
  } catch(SAXParseException ex){ // construct error message
     outputString="<?xml version='1.0' encoding='utf-8'?>\n"+ "<error>\n"+
                  " <src>"+ex.getSystemId()+"</src>\n"+
                  " <line>"+  ex.getLineNumber()+"</line>\n"+
                  " <pos>"+   ex.getColumnNumber()+"</pos>\n"+
                  " <reason>"+ ex.getMessage()+"</reason>\n"+ "</error>";
  } catch(Exception ex){outputString="<error>"+ex.getMessage()+"</error>";}
%><%= outputString %>
```

DTD Syntax and Examples

Now that we can test and experiment, here is a systematic overview of DTD syntax, beginning with element declarations and content models.

Element Declarations and Content Models

Element declarations have the following form:

```
<!ELEMENT element-name (content-model) >
```

For example:

```
<!ELEMENT exchange (q,a)>
```

An element's name must be a legitimate namespace-aware name as specified in the qName production of *XML Namespaces.* (See the section "Syntax of Names and EBNF Productions" in Chapter 2.) The content model defines the internal syntax of the element. Five content models can be distinguished: ANY, EMPTY, children-only, text-only, and mixed (text and children).

ANY and EMPTY

The ANY content model places no restriction on the element's content. It is not very useful because everything within that content still has to be declared. For instance, if you declare

```
<!ELEMENT anyContent ANY>
```

and, then, in your document instance, write

```
<anyContent>
  <child>c</child><orphan>o</orphan><waywardChild>w</waywardChild>
</anyContent>
```

you still have to declare `child`, `orphan`, and `waywardChild` in the DTD for your document to be valid. By contrast, recent alternatives to DTD—XML Schema and RELAX NG—allow element declarations that really place no restrictions on the element's content in document instances. Such elements can be given any well-formed content in an instance document, and the document will remain valid (assuming that the rest of it is valid).

The EMPTY content model is for empty elements, such as `<br/>` or `<img src="img.jpg"/>`.

Children-Only

The children-only content model allows combinations of choice and sequence "content particles" that can be of arbitrary depth. The comma is the sequence separator, and the pipe character (|) is the choice separator. In addition, the familiar single-character operators (?, *, and +) indicate how many times an element, a sequence, or a choice may appear. Exactly the same operators and the pipe character (but not the comma) are used in the EBNF productions that define the syntax of DTD declarations. So we have a meta-language (EBNF) that shares some vocabulary with its object language (DTD), which is itself a meta-language for XML languages.

> **NOTE** EBNF, *first introduced in Chapter 2, stands for* Extended Backus-Naur Form.

Here are the EBNF productions. They come from two W3C recommendations: *XML 1.0* (46, 47, 49, and 50) and *XML Namespaces* (14 and 15). The *XML Namespace* productions supercede productions 45 and 50 of *XML 1.0*.

For readability, we have removed multiple occurrences of S? that indicate where optional whitespace may appear. The cp symbol stands for "content particle".

```
[14]  elementdecl ::=  '<!ELEMENT' S QName S contentspec '>'
[46]  contentspec ::=  'EMPTY' | 'ANY' | Mixed | children
[47]  children ::=  (choice | seq) ('?' | '*' | '+')?
[15]  cp ::=  (QName | choice | seq) ('?' | '*' | '+')?
[49]  choice ::=  '(' cp ('|' cp)* ')'
[50]  seq ::=  '(' cp (',' cp )* ')'
```

Here are some examples of children-only content models.

```
<!ELEMENT p (a,b,c)>
<!ELEMENT p ((a|b),c,d*)>
<!ELEMENT p ((a|b),c?,d*,((e|f),g)+)>
```

The last example says the content model of the `p` element is `a` or `b`, followed by an optional `c`, followed by 0 or more `d`'s, followed by 1 or more of (`e` or `f` followed by `g`).

Mixed and Text-Only

The mixed-content model must be specified as a choice whose first element is #PCDATA, repeated an arbitrary number of times. Here is the EBNF production.

```
[16]  Mixed ::=  '(#PCDATA' ('|' QName)* ')*' | '(#PCDATA)'
```

Here is an example of a declaration and a matching element within a document instance.

```
<!ELEMENT mix ((#PCDATA|em|strong)*)>
<mix>Text with <em>emphasis</em> and some <strong>boldness</strong>, too.</mix>
```

A mixed-content declaration says, in effect, to repeat #PCDATA or the other possible choices any number of times in any order. The consequence of this way of declaring mixed content is that it is impossible to specify the order of children elements within the text or to make some elements required. If, for instance, you want to allow freeform bug reports with markup for system specification and program version, the DTD gives you no way to ensure that those elements always appear in a specific order. By contrast, both XML Schema and RELAX NG allow you to express these constraints.

If you review productions 46 and 16, you will see that *XML 1.0* treats text-only as a special case of mixed, with the number of children equal to 0. Formally, it makes sense, but the usage is different enough to justify a distinction: the text-only content model is common in data-oriented XML in leaf elements of a regularly structured tree, whereas the mixed-content model is characteristic of document-oriented XML, in which markup is commonly inserted in long stretches of text.

Attribute Declarations

Attribute declarations have the following format (this time we'll start with EBNF):

```
[17]  AttlistDecl ::=  '<!ATTLIST' S QName AttDef*  '>'
[18]  AttDef ::=  S (QName | NSAttName) S AttType S DefaultDecl
```

Production 17 says that an attribute declaration consists of a constant string `<!ATTLIST` followed by a qName (the name of the element that owns the attributes) and any number of attribute definitions. An attribute definition starts with a name (a qName for a real attribute, an `xmlns:` name for a namespace declaration) and proceeds with a type and a default declaration. The "default declaration" (`DefaultDecl` in production 18) can indeed be the default value of the attribute, or it can be one of three constants: `#REQUIRED`, `#IMPLIED`, `#FIXED`. Their meaning is that the attribute is required, optional, or has a fixed value, respectively.

Listing 3-4 provides an example.

Listing 3-4. Attribute Declarations

```
<!ATTLIST someElement
  name      CDATA         #REQUIRED
  ISBN      ID            #REQUIRED
  ref2id    IDREF         #IMPLIED
  penname   NMTOKEN       #IMPLIED
  authors   NMTOKENS      #REQUIRED
  answer    (YES|NO)      "NO"
  dessert CDATA "strawberries with icecream"
  method    CDATA         #FIXED "GET"
>
```

As you can see, the value of an attribute can be of several different types (unlike the text content of an element, which can be of only one possible type, PCDATA). The most general attribute type is CDATA (Character DATA). It is not PCDATA because attribute values are quoted and thus not parsed. However, to make the parser's life easier, you still have to encode the < and > characters as `<` and `>`.

In addition to CDATA, an attribute value can be

- NMTOKEN: a string that matches the Name production; no whitespace
- ID: a NMTOKEN that is unique within a document. An attribute of type ID doesn't have to be named `id`, as Listing 3-4 shows.
- IDREF: the value of an attribute of type IDREF must be the value of an attribute of type ID in the same document. This is a validity constraint that is checked by a validating parser.
- NMTOKENS, IDREFS: whitespace-separated lists of NMTOKEN and IDREF

Finally, an attribute type can be a vertical-bar-separated enumeration of possible values, as in `YES|NO` in Listing 3-4.

Attribute Defaults

The third item in an attribute declaration is a mixed bag of possibilities. It can be one of two literals, #REQUIRED or #IMPLIED, the second meaning "optional." Alternatively, it can provide a default value. This is perhaps the most controversial feature of DTDs because it has nothing to do with validation: instead of constraining the document, the DTD adds information to it. Many think that this is a bad idea, both philosophically (because it's not the DTD's business to add information to the document) and practically, because the document will be different depending on whether the parser has access to the DTD and is configured to consult it. This is not a problem in SGML for which the DTD is always available and needed, but attribute defaults are problematic in XML.

The behavior of the parser with respect to attribute defaults is not completely specified. If the parser is validating and validation is on, then, of course, the default value will be supplied; if the validation is not on or the parser is not a validating parser at all, then it is not required to consult the DTD, but most of them do anyway. The parser in Internet Explorer does not validate but supplies default attribute values.

If the default is the only possible value, you can insert the #FIXED keyword before it. This is a minor optimization feature.

Infoset Augmentations

In technical language, if a DTD adds an attribute default to its document instance, we say that it "augments the document's infoset" by adding an attribute information item. In addition to default attributes, DTDs can change the document's infoset by modifying whitespace. Suppose, you have an attribute as follows:

```
<someElement attr="   words     with    whitespace   "
```

If the DTD says that the attribute type is CDATA, all whitespace should be preserved. If the attribute type is NMTOKENS, then the whitespace should be normalized: leading and trailing whitespace removed, and all other sequences of whitespace characters replaced by a single space character, #x20. A similar distinction applies to an element that contains children elements and whitespace: if the element's content model is children-only, the whitespace should be removed; if it's mixed (text and children), then the whitespace should be preserved.

RELAX NG and XML Schema have diametrically opposite attitudes to infoset augmentation: RELAX NG avoids it completely whereas XML Schema augments the infoset much more aggressively than does the DTD. A special term,

Post-Schema-Validation Infoset (or *PSVI*) has been coined for the result of XML Schema infoset augmentations.

Consistently minimalist, RELAX NG does not even specify how an XML document is to refer to its RELAX NG grammar. Both the *XML 1.0* and *XML Schema* recommendations do provide such specifications.

A Document and Its DTD

A DTD can be internal, external, or consist of an internal and an external subset. Our examples so far have shown internal DTDs. External DTD references are given as part of the document type declaration, as follows:

```
<!DOCTYPE tstmt SYSTEM "../common/tstmt.dtd">
<!DOCTYPE wml SYSTEM "http://www.wapforum.org/DTD/wml12.dtd">
```

As you can see, a DTD reference is a relative or absolute URI that is preceded by the SYSTEM keyword. Parts of the document content can also be placed in an external file and referred to in the same way. Such included external material is called an *external general parsed entity*. External entities other than the DTD have to be named and referred to by name; external DTDs are, in effect, unnamed external entities. We will show external DTDs and DTDs containing both an internal and an external subset after we discuss entities.

General and Parameter Entities

Entities are physical units such as characters, strings, and files. Entity references are syntactic constructs that refer to those units. Entity declarations associate a name with an entity. An entity reference is formed from an entity name by prefixing it with & (for character entities and general entities) or `%` (for parameter entities) and adding a semicolon in the end. See Table 3-1.

Table 3-1. Entity, Entity Name, and Entity Reference

ENTITY	ENTITY NAME	ENTITY REFERENCE
<	lt (pre-declared)	<

Parameter entity references can appear only in the DTD, serving, for instance, as a macro name. General entity references are mostly intended for use in the document, although they can appear in the DTD as well. In Chapter 2, we saw the five predeclared general entity names: `lt`, `gt`, `quot`, `apos`, and `amp`. All

other general entity names, including those that are predeclared in most HTML parsers (such as `nbsp`) have to be declared in the DTD before they can be used in an XML document.

General entities can be parsed or unparsed. General parsed entities are XML data (perhaps with no markup), whereas general unparsed entities are non-XML data: either binary data such as a JPEG image or non-XML character data such as LaTeX. General unparsed entities have to be external files referenced by a URI. General parsed entities can be external or internal. Parameter entities can also be internal or external. Here is a summary list of entities, followed by examples.

- character entities
- five predeclared single-character general parsed entities
- parsed general entities: internal/external (XML)
- unparsed general entities, external only, non-XML data; must be accompanied by a NOTATION declaration to describe the format of the data
- parameter entities, internal/external

The following examples show parsed general entities and parameter entities. Unparsed general entities are rarely used, and we will leave them alone, together with notation declarations. To include unparsed material, you can use, for instance, XLink and the material's MIME type to indicate how it should be processed by your application.

Parsed General Entities

The syntax for declaring general entities is as follows:

```
<!ENTITY name "string value or URI reference">
```

For example (divided into two lines):

```
<!ENTITY standardDisclaimer
    "<disclaimer>We are not liable for anything ever.</disclaimer> ">
```

The next example shows two general parsed internal entity references: one within the DTD itself, the other in the document. There is also a character entity reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE r [
 <!ELEMENT r (c+)><!–r has one or more c children -->
 <!ELEMENT c (#PCDATA|d)*><!–c has "mixed" content model -->
 <!ELEMENT d (#PCDATA)><!–d is text only, a leaf node -->
 <!ENTITY rights "All rights reserved.">
<!--entity can contain markup and include other entities by reference,
        including character references and previously declared entities -->
 <!ENTITY standardDisclaimer
     "<d>We are not liable for anything ever.
Disclaimer &#xA9; 1998. &rights;</d> ">
]>
<r><c>
  As always, our standard disclaimer applies: &standardDisclaimer;
</c></r>
```

With the default IE stylesheet, this comes out as shown in Figure 3-3.

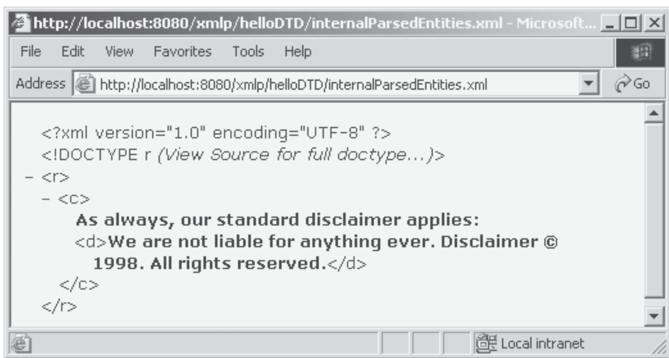

Figure 3-3. Internal general entities

The process of replacing an entity with its value is called *entity resolution*. Two internal general parsed entities are declared in the DTD: `rights` and `standardDisclaimer`. The first of them is referenced within the DTD itself as part of the declaration of the second one. Both the reference to `rights` and the character entity reference are resolved as part of the second entity declaration. (Even though the entity value is quoted, it is parsed and entity references within it are resolved.) The second entity reference is in the document itself. The entity can contain markup, but it must be balanced: an element cannot start in one entity and end in another.

External General Parsed Entities

General entities can be external to the document, referenced by an absolute or relative URI. We can modify our example by placing the text of the references into separate files in the entities subdirectory and rewriting the declarations as shown in Listing 3-5.

Listing 3-5. DTD with External Parsed Entities

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE r [
 <!ELEMENT r (c+)>
 <!ELEMENT c (#PCDATA|d)*>
 <!ELEMENT d (#PCDATA)>
 <!ENTITY company "N-topus &#38;#38; son.">
<!-- to output the & character in the document instance,
       we have to escape it twice:
       the first escape is removed while processing the DTD,
       the second is removed in parsing the document
 -->
<!-- the next two declarations use external entities, specified by URIs preceded
by the keyword SYSTEM.
The first is specified by an absolute URI, the second by a relative URI -->
 <!ENTITY rights SYSTEM
    "http://localhost:8080/xslp/helloDTD/entities/rights.txt">
 <!ENTITY standardDisclaimer SYSTEM "entities/stdd.xml">
]>
<r> <c>Our standard disclaimer applies: &standardDisclaimer;</c>
        Your rights are as follows: &rights;
</r>
```

To have external entities resolved correctly, the document has to be viewed over HTTP, not as a local file. It comes out as shown in Figure 3-4.

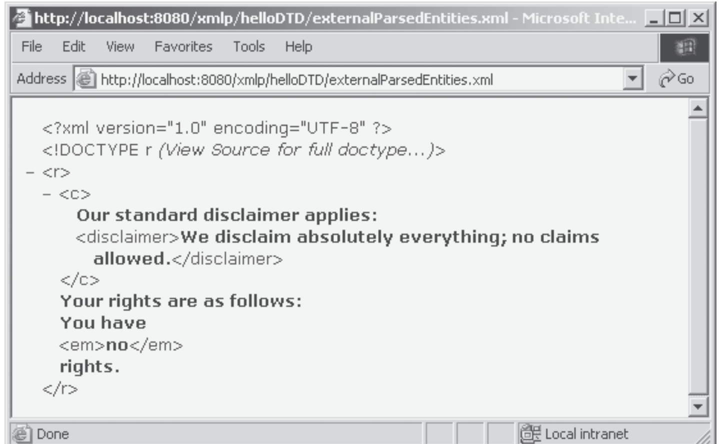

Figure 3-4. External general entities

Now that we have introduced and provided examples of external entities, let's look in detail at how they are referenced.

External Entity References (Including DTD References)

The external entity declarations in our example are system declarations: they consist of the keyword SYSTEM followed by an absolute or relative URI:

```
<!ENTITY rights SYSTEM
  "http://localhost:8080/xmlp/helloDTD/entities/rights.txt">
 <!ENTITY standardDisclaimer SYSTEM "entities/stdd.xml">
```

The same syntax is used for external DTDs, which are, in effect, external entities without a name:

```
<!DOCTYPE tstmt SYSTEM "../common/tstmt.dtd">
<!DOCTYPE wml SYSTEM "http://www.wapforum.org/DTD/wml12.dtd">
```

System declarations identify an entity or DTD by its URI, which is its absolute or relative address on the Web. This is not a very good way to identify entities because Web addresses change and sometimes disappear, and the same entity may be found at different addresses. With much-used DTDs such as XHTML or WML (Wireless Markup Language), it is a common practice to validate using a local copy, but SYSTEM identifiers do not provide any way to link your local copy to the authoritative one. In the world of SGML, there were public identifiers, preceded by the PUBLIC keyword, that provided that functionality via

locally maintained registries. SGML public identifiers follow a precisely defined syntax and are technically known as *Formal Public Identifiers* (*FPIs*). The FPI syntax is as follows. A public identifier consists of four fields separated by the "//" delimiter:

- The first field is "ISO" for ISO-approved documents, "+" for documents approved by some other standards body, and "–" for everything else. (W3C uses "–" for itself because it is not a standards body but rather an industry consortium.)
- The second and third are owner and title fields, which contain text.
- The last field is the language code, a two-letter string specified in the ISO-639 standard. (This standard is also used elsewhere in XML, as values of the `lang` and `xml:lang` attributes, for instance.)

XML inherited SGML public identifiers, and many well-known DTDs have them, as in

```
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
"http://www.wapforum.org/DTD/wml12.dtd">
```

As you can see, the PUBLIC identifier is here followed by a SYSTEM identifier, without the SYSTEM keyword. This is actually a requirement specified in *XML 1.0* because, in the absence of an established and standard registry structure, PUBLIC identifiers are not very useful. However, there is an effort at OASIS to develop a standard for a network of authoritative registries, similar in their function to domain name servers. In anticipation, many DTDs start with a comment that spells out the recommended usage. This is especially true about DTDs produced by large organizations and consortia, such as W3C. Here is an example from WML DTD. (As you may know, WML is a display language for mobile devices. It is part of WAP, the Wireless Application Protocol, developed by WAPForum [`www.wapforum.org`]. Later in the chapter, we will show the DTD for XHTML Basic, which will probably replace WML within the next few years.)

```
<!--
        Wireless Markup Language (WML) Document Type Definition.
        WML is an XML language. Typical usage:
        <?xml version="1.0"?>
        <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
        "http://www.wapforum.org/DTD/wml12.dtd">
        <wml>
```

```
        ...
        </wml>
        Terms and conditions of use...
-->
```

The WML public identifier follows the syntax of SGML's Formal Public Identifiers, out of respect for SGML legacy and for possible future use. An OASIS technical committee is working on a system of Internet-wide registries for public identifiers, tentatively called *XML catalogs*.

What Are General Entities Good For?

Internal general parsed entities are, in effect, macros or global constants. Like macros or global constants, they are useful as a shorthand for repeating material. It is not clear that putting them in a DTD is a good idea because, if the DTD is not read, the references will remain unresolved. It is probably better to store the repeating material as XML and include it using XLink/XPointer. (An XInclude specification is also working its way through the pipeline, although it is still in a very tentative stage.)

External general entities are for including boilerplate material or creating modular documents. It certainly feels good to be able to have a document like this:

```
<ns:book>&frontMatter; &intro; &ch01; &ch02; &concs; </ns:book>
```

External general entities are obviously replaceable by XLinks. The goal is to reference some resource using a local name that is mapped to a URI; declaring the mapping directly in the document is better than putting the entity definition in a grammar that may or may not be available. In the case of unparsed entities, this is pretty much the consensus: to include unparsed content, use an XLink or have the application process the URI declared as an attribute:

```
<image src="images/anImage.jpg"/>
```

Neither XML Schema nor RELAX NG has anything analogous to general entities, and, if you are not an SGML pro who is addicted to them, it may be wise to stay away.

Parameter Entities

Parameter entities (PEs) do have analogs in both XML Schema and RELAX NG but with a more narrow and focused range of uses. In this section, we present the syntax and give simple examples. More-complex uses of PEs having to do with DTD modification and reuse will follow in a separate section.

PE Declaration and Reference; PEs as Macros

Both the declaration and reference contain the percentage (%) symbol. In the declaration, a space must separate the percentage symbol from the entity's name. This example is an excerpt from WML1.2 DTD:

```
<!-- Task types -->
<!ENTITY % task "go | prev | noop | refresh">
...
<!ELEMENT do (%task;)>
<!ELEMENT onevent (%task;)>
```

This example shows a PE whose value is a content model for two different elements. Instead of repeating it twice, a PE is declared and used.

In a similar way, the XHTML DTD declares a PE called `coreattrs` that is used in attribute declarations for many elements:

```
<!ENTITY % coreattrs
 "id          ID              #IMPLIED
  class       CDATA           #IMPLIED
  style       %StyleSheet;    #IMPLIED
  title       %Text;          #IMPLIED"
  >
<!ATTLIST br
  %coreattrs;
  >
```

PEs as Documentation

WML1.2 DTD opens with these three declarations:

```
<!ENTITY % length "CDATA">
    <!-- [0-9]+ for pixels or [0-9]+"%" for percentage length -->
<!ENTITY % vdata "CDATA">
    <!-- attribute value possibly containing variable references -->
```

```
<!ENTITY % HREF "%vdata;">
    <!-- URI, URL or URN designating a hypertext node.

           May contain variable references -->
```

All three declare synonyms for CDATA, a quoted character string used as an attribute value. Their only purpose is to document the intended meaning of the attribute; for instance, in declaring the `img` element's attributes, the WML DTD says:

```
<!ATTLIST img
  alt %vdata; #REQUIRED
  src %HREF; #REQUIRED
  localsrc %vdata; #IMPLIED
  vspace %length; "0"
  hspace %length; "0" . . . >
```

This is better than declaring `src`, `vspace`, and `hspace` as CDATA, but, of course, DTD validation does not enforce the intended meaning by checking that `%HREF;` is indeed syntactically a URL or that `%length;` is a length specified as an integer or a percentage. Similarly, the following two declarations are just syntactical sugar for human readers:

```
<!ENTITY % boolean "(true|false)">
<!ENTITY % number "NMTOKEN"> <!-- a number, with format [0-9]+ -->
```

Virtually all large, production-quality DTDs from W3C or industry groups contain multiple declarations of this sort. These are unnecessary in XML Schema and RELAX NG in which an extensive system of data types is available.

PEs are heavily involved in DTD modification, modularization, and reuse. This is because, in a DTD, you cannot redefine an element or an attribute, but you can redefine a PE. The syntax of DTD modularization and reuse is quite complex. We will show it, with examples, in a separate section. Our immediate task is to bring together everything that can be found in a DTD and a (serialized) XML document. After that, the *Infoset* specification will make sense, and we will take a brief look at it. (As you recall, the *Infoset* recommendation is intended to harmonize tree models used in other recommendations, such as *DOM*, *XPath*, *XPointer*, and (forthcoming) *XML Query Language*.)

DTD Modification and Reuse

To have an example to experiment with, we'll write down a DTD for the pdata (personal data) documents of Chapter 1. Listing 3-6 shows the simplest version.

Listing 3-6. A DTD Example, pdata0.dtd

```
<!ELEMENT pdata (person*)>
<!ELEMENT person (name,address,bdate,email,favorites)>
<!ATTLIST person
  id          ID    #REQUIRED
  access-level CDATA #REQUIRED
>
<!ELEMENT name (title?,last,first,middle?)>
<!ELEMENT address (street,city,state,zip)>
<!ELEMENT bdate (year,month,day)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT favorites (color,drink)>
<!-- everything else is #PCDATA: title, first, last, middle, street, city, etc -->
```

We can use this DTD to validate pdata documents of Chapter 1. We can also use to it to validate documents that consist of a single person element or a single address element. In other words, the document type doesn't have to be the maximal element declared in the DTD; this document will validate correctly:

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "../dtd/pdata0.dtd">
 <person id="Prof123" access-level="teacher"><!-- one person's data --></person>
```

Reuse with an Internal Subset in a Document

Suppose we have to deal with different kinds of persons who have different sets of favorites. Here are the basic facts:

- You cannot redefine elements or attributes, but you can redefine PEs.
- If a PE has more than one definition, the first one wins and the rest are ignored.
- If a DTD has an internal and an external subset, the internal subset is processed first, so its definitions override those in the external subset.

So, in order to reuse a DTD, first we go through the following steps:

1. Declare what needs to be modified (for example, an element's content model) as a PE.

2. Make the DTD an external subset.

3. Redefine the PE in the internal subset.

In our case (see pdata.dtd), we replace the line

```
<!ELEMENT favorites (color,drink)>
```

with

```
<!ENTITY % favoritesContModel "color,drink">
<!ELEMENT favorites (%favoritesContModel;)>
```

This will still validate all those people who have a favorite drink. For a person who has a favorite book instead, we proceed as shown in Listing 3-7.

Listing 3-7. Overriding Content Models

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "../dtd/pdata.dtd" [
<!ENTITY % favoritesContModel "color,book">
<!ELEMENT book (#PCDATA)>
]>
  <person id="Prof123" access-level="teacher">
  ...
    <favorites>
      <color>red</color>
      <book>Wisdom Book</book>
    </favorites>
  </person>
```

This will validate correctly.

Validating Namespaced Documents

A similar technique can be used to validate qNames in documents with namespaces. Remember that DTDs predate *XML Namespaces* and have the *XML 1.0* view of names. This results in two problems.

- For DTDs, a namespace declaration is just an attribute and needs to be declared as such.
- If the namespace declaration is declared as an attribute with a specific prefix mapping and you later change the prefix, the document will no longer be DTD valid.

Certain risks are associated with declaring a specific prefix in the DTD; for instance, parts of your document may be pasted into a different document where they may be in the scope of a different namespace declaration with the same prefix. If it is safe to hardwire the prefix into the DTD, then the `xmlns` attribute should be declared `FIXED`. If such an attribute has a value that is different from its fixed value, the validating parser will catch that as a parsing error. (Some namespace-aware parsers insist that `xmlns` attributes are declared `FIXED`.)

If you want to preserve the flexibility of changing the prefix mapping in individual documents, declare two PEs: `suff` and `pref`, one for the suffix that follows the string `xmlns` and the other for the prefix. For documents with a different prefix, you redefine `suff` and `pref` in the internal subset. So, for instance, your nsdtdexample.dtd might look as shown in Listing 3-8.

Listing 3-8. Namespaces and DTDs

```
<!ENTITY % pref "" >
<!ENTITY % suff "" >
<!ENTITY % example.qname "%pref;example" >
<!ENTITY % nsdecl "xmlns%suff;" >
<!ELEMENT %example.qname; (#PCDATA) >
<!ATTLIST %example.qname;
  %nsdecl; CDATA #IMPLIED
>
```

This will validate a document that uses a default namespace. To validate a document that has the namespace URI mapped to a prefix, redefine `suff` and `pref` in the internal subset, as shown in Listing 3-9.

Listing 3-9. Namespace Suffix Redefined in the Internal Subset of DTD

```
<!DOCTYPE example SYSTEM "nsdtdexample.dtd" [
<!ENTITY % pref "a:" >
<!ENTITY % suff ":a" >
]>
<a:example xmlns:a="http://my.namespace.location.net/example">
  Hello, Namespace!
</a:example>
```

This technique is used in several W3C DTDs, including *Modularized XHTML* and *MathML*.

Reuse Within Another DTD

We can also reuse a DTD within another DTD: for people who like books, we produce a separate DTD that looks like this (profdata.dtd):

```
<!ENTITY % favoritesContModel "color,book">
<!ENTITY % pdatadtd SYSTEM "pdata.dtd"><!—make the old DTD a PE —>
%pdatadtd; <!— include it by reference —>
<!ELEMENT book (#PCDATA)><!— add new element(s) as needed —>
```

The new DTD includes the original DTD as a PE reference and adds declarations for the new element.

Elements vs. Attributes and Attribute Modifications

Listing 3-10 is an example of the color element redefined as EMPTY and color values placed in an attribute (pdatac.dtd).

Listing 3-10. An Element Redefined

```
<!ELEMENT email (#PCDATA)>
<!ENTITY % favoritesContModel "color,drink">
<!ELEMENT favorites (%favoritesContModel;)>
<!-- name, address and date elements declared here -->
<!ELEMENT color EMPTY>
<!ENTITY % mincolors "red|blue|green">
<!ENTITY % morecolors "">
<!ENTITY % colorattr "favcolor (%mincolors;%morecolors;) #REQUIRED">
<!ATTLIST color
  %colorattr;
>
<!ELEMENT drink (#PCDATA)>
```

This validates the following document:

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "../dtd/pdatac.dtd">
<person id="Prof123" access-level="teacher">
```

```
 <!-- person data: name, address, etc -->
<favorites>
      <color favcolor="red"/>
      <drink>cookie juice</drink>
    </favorites>
</person>
```

To add more colors, we redefine `morecolors`, as before.

```
<!ENTITY % favoritesContModel "color,book">
<!ENTITY % morecolors "|maroon|cyan">
<!ENTITY % pdatadtd SYSTEM "pdatac.dtd">
%pdatadtd;
<!ELEMENT book (#PCDATA)>
```

More favorite colors are now available. Note that, to be reusable, a DTD must contain a "hook," that is, a parameter entity that can be redefined. Note also that the redefined `morecolors` PE has to start with the pipe character (`|`). In general, the reuse is not done by reusing structure but by modifying strings of characters, because that is what the value of a PE is: a string of characters. A system of code reuse that is all built around macros and character strings is bound to be awkward to use, prone to error, and difficult to maintain. This is the primary reason why it makes sense to switch to an XML language for XML grammars: not because XML syntax is better for human users (DTDs are quite compact and easy to read), but because its content models and attribute values become structured entities that are easy to work with.

Conditional Sections

Conditional sections make it possible to include some portions of the DTD conditionally, depending on the value of the conditional section's keywords. The keywords are INCLUDE and IGNORE. In the following code, if we replace `INCLUDE` with `IGNORE`, the editorial comment element will no longer be defined.

```
<![INCLUDE [
  <!ELEMENT proofreaderComment (#PCDATA)>
]]>
```

As you can see, conditional sections are syntactically similar to CDATA sections but appear in the DTD and not in the document.

How is this useful? If you want something included, why can't you just include it? And, if you want something ignored, why can't you simply comment it

out? Conditional sections are useful because the keywords can be defined (and redefined) as PEs:

```
<!ENTITY % editing "INCLUDE">
<![%editing; [
  <!ELEMENT proofreaderComment (#PCDATA)>
]]>
```

Once the editorial process is over, redefine `editing` as `IGNORE`, and the validating parser will throw an error if there are any `proofreaderComment` elements left in the document (because the element is no longer defined). Because `editing` is a PE, it can be redefined in the internal subset within a document instance, without changing the DTD.

You can define the same element in two conditional sections, provided that they are not both included at the same time. In this example, the `q` element's content model within `practiceExam` includes answers whereas it doesn't for `realExam`:

```
<!ENTITY %  practiceExam  'INCLUDE' >
<!ENTITY %  realExam 'IGNORE' >
<![%practiceExam;[<!ELEMENT q (text, hint*, answer)> ]]>
<![%realExam;[<!ELEMENT q (text, hint*)> ]]>
```

For a practice exam, we declare PEs as before. For a real exam, we switch them around.

XHTML Modularization and XHTML Basic

XHTML modularization is central to client-side display of XML. The vision has always been to remain with the broadly known vocabulary of HTML but to divide it into small, independently reusable modules that each provide specific functionality. Modules can be combined into precisely targeted languages that would require much smaller parsers to validate than would the entire XHTML. (Without some such capability, XHTML could not be used on small devices, and there was a danger that the wired and wireless Webs would have difficulty communicating.) The module mechanism can be used to create new modules that are easy to combine with XHTML modules. Primary candidates for such new modules would be MathML (to display mathematical expressions) and SVG (Scalar Vector Graphics) to add XML-based graphics to Web pages.

It was fairly obvious from the beginning that implementing a modularization framework as DTDs is possible but awkward. Part of the original vision was to develop an XML Schema language that would be much better suited for modularization and reuse. That part of the vision failed. XML Schema took much

longer than expected to develop. As its details started taking final shape, it turned out, somewhat embarrassingly, that it was not very well suited for XHTML modularization. Because its release could no longer be delayed due to the pent-up pressure of other projects that were dependent upon it, the entire sequence of XHTML specifications was released with DTD implementations only. They include, in reverse chronological order:

- *XHTML 1.1—Module-Based XHTML* (May 31, 2001)
- *Ruby Annotation* (May 31, 2001)
- *Modularization of XHTML* (April 10, 2001)
- `Mathematical Markup Language (MathML)` Version 2.0 (February 21, 2001) (implements MathML as a module)
- *XHTML Basic* (December 19, 2000) (depends on *Modularization of XHTML* and *XHTML 1.1*)

Modularization of XHTML defines the overall framework for modularization. *Ruby Annotation* defines a small module for "short runs of text alongside the base text, used in East Asian documents to indicate pronunciation or to provide a short annotation." *XHTML 1.1* defines the XHTML document type as a collection of modules implemented as a DTD. *XHTML Basic* is a subset of *XHTML 1.1* defined within the modularization framework. *MathML Version 2.0* defines the MathML DTD as a module and provides a combined MathML+XHTML DTD.

> **NOTE** *As of this writing, both* Modularization of XHTML *and* XHTML Basic *have empty appendices promising Schema Module Implementations.* XHTML 1.1 *has been released without such an appendix. There is a working draft of an implementation published in March 2001 at* `www.w3.org/TR/xhtml-m12n-schema/`. *(Note also the newly fashionable shortening of* modularization *to* m12n*, with the number 12 representing the twelve internal letters that are thus omitted. The first instance of this odd notational shorthand, as far as we know, was* i18n*, for* internationalization*, used in the HTML DTD recommendation.)*

XHTML Basic and Wireless Devices

You will notice that *XHTML Basic* was released before the two recommendations on which it depends. This is because the release of *XHTML Basic* could not be delayed, either. A display language for small devices was one of the main motivations for the modularization framework. XML was from the beginning created with small devices in mind; *XML 1.0* mentions them as one of the reasons why XML documents should be parsable without a grammar. To preserve the unity of the Web, it was clearly desirable to give them a display language that is a proper subset of XHTML.

Developing a general modularization framework using DTDs took a long time, and developing XML Schema took even longer. In the meantime, two other display languages for mobile devices were spreading rapidly: WML, developed by WAP Forum, and cHTML (Compact HTML) that was adopted, with extensions, by NTT's DoCoMo in their iMode wireless service. The Web seemed in danger of splintering into the wired and wireless domains, but the danger was averted after W3C released *XHTML Basic*. Both WAP Forum and NTT (in alliance with AOL) quickly expressed public support for it. WML and cHTML are likely to persist until devices that support them are phased out, which may take a few years.

Background: Evolution of HTML and XHTML

HTML 3.2 was to a great extent a codification of existing practice, which included many questionable features. Particularly offensive to the spirit of SGML and XML were purely presentational tags, such as `center` or `font`, that violated the principle of separating structure from presentation. (Presentation must be provided by a stylesheet.) HTML 4.0 contains four DTDs: strict, transitional, lax, and frames. The first three are alternative definitions of head-body documents that differ in their degree of tolerance for "legacy" HTML: the strict DTD prohibits legacy elements altogether, the transitional one deprecates them, and the lax one lets them all in. The frames DTD defines head-frameset documents. XHTML 1.0 drops the legacy DTD but otherwise is a straight XML rewrite of HTML 4.0. XHTML 1.1 is a modularized reformulation of XHTML 1.0 strict.

> **NOTE** *A separate frames DTD is a historical relic. (See James Clark's discussion in* `http://lists.w3.org/Archives/Public/www-html-editor/2001JanMar/0039.html`*). It could be quite simply integrated in the other DTDs by redefining the content model of the HTML element as (head, (body|frameset)). For some reason, neither XHTML 1.0 nor 1.1 chose to make the change.*

Modularization Goals and Use Cases

The goal of modularization is to support the following use cases:

- *subset*: Specify a language that is a subset of XHTML. XHTML Basic is an example.
- *extend*: Add elements and/or attributes to XHTML or its subset. RDDL is an example: it adds a single element and several XLink attributes to XHTML.
- *embed*: Reuse (parts of) XHTML vocabulary within another language.

To support these use cases and ensure interoperability, XHTML modularization breaks the large vocabulary of HTML into small modules. The operation of subsetting is done on modules and not individual elements or attributes. To add another vocabulary, such as MathML, it is also packaged as a module and added in the same way that XHTML modules are added to a subset of XHTML.

Abstract Modules and Implementations

The framework is organized in two layers: abstract modules, each providing specific functionality, and implementations of those modules (currently, DTD only). The modules are divided into three groups, as follows:

The first group consists of a single module that defines common sets of attributes used by many elements in different modules; it also contains definitions of attribute data types, such as NMTOKEN.

The core group of modules contains modules that are required to be present in any document type that conforms to the XHTML family. The idea is to ensure a certain level of interoperability even between different XHTML-based languages. The core group contains four modules: structure, text, hypertext, and list. The structure module defines the following elements: `html`, `head`, `title`, and `body`. The other three modules of the core group support specific functionalities, as their names suggest.

The rest of the modules, too numerous to list here, are also functionality specific, with names such as Applet, Image, Client-side Image Map, and so on. Forms and tables modules, in addition to regular (X)HTML versions, have reduced versions for XHTML Basic.

> **NOTE** *A complete list of modules can be found at* `www.w3.org/TR/xhtml-modularization/abstract_modules.html#s_xhtmlmodules`.

Using an XHTML language without validation is trivial: just use the vocabularies you want. Validation, however, is another story: it requires a careful "packaging" of modules so that they can be reused and extended in other grammars. In other words, abstract modules have to be implemented in a grammar formalism in a way that supports the use cases. The remainder of this chapter shows the DTD implementation, released by W3C as part of *XHTML Basic*.

The Framework

The DTD implementation of the modularization framework is based on extensive use of PEs, especially conditional sections and the techniques of reuse that we have seen in the last two sections. It mostly consists of files that implement shared material, such as the definitions of core attributes and their data types. These files have a .mod extension. A "qname" file contains the machinery to support namespaces. The "framework" file includes (using conditional sections) all the general-purpose modules and also the required modules that all XHTML-based languages must include.

PE Naming Conventions

The framework is not easy to use. To make it more manageable, we are given an elaborate system of PE naming conventions, both for modules themselves and for their content. Consider the list module for creating definition lists, ordered (numbered) lists and unordered (bulleted) lists. It contains these elements: `dl`, `dt`, `dd`, `ol`, `ul`, and `li`. Supporting the module are two PEs named `xhtml-list.module` and `xhtml-list.mod`. The `.module` PE is declared to be INCLUDE or IGNORE and used as the keyword of a conditional section. The `.mod` PE is declared to be the URI of the file that contains the actual declarations. (All such files have names with the .mod extension.) In the DTD, the MOD file is included by reference, within a conditional section controlled by the value of the `.module` PE, as shown in Listing 3-11.

Listing 3-11. Example of `.module` *and* `.mod` *PEs*

```
<!-- Lists Module (required) ....................................-->
<!ENTITY % xhtml-list.module "INCLUDE" >
<![%xhtml-list.module;[
<!ENTITY % xhtml-list.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Lists 1.0//EN"
            "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-list-1.mod" >
%xhtml-list.mod;
]]>
```

Each XHTML and XHTML Basic module has such a `.module` PE and a `.mod` PE. In Listing 3-11 (which is quoted from the W3C recommendation), `xhtml-list.mod` is defined as a URI pointing to the xhtml-list-1.mod file on the W3C site. It is common practice to have a local copy and use a relative URI. It becomes your responsibility to update it when a new version comes out.

PE Naming Conventions Within a Module

Within modules are the following standard PE names for individual elements and attributes and for groups of elements or attributes:

- `.qname`: to deal with namespace issues (`qname` PEs contain `prefix` and `suffix` PEs as in our Listing 3-8 and 3-9 in the section on DTD validation of namespaced documents)
- `.content`: to represent the content model of an element type
- `.class`: to represent elements of similar meaning (the W3C recommendation calls them "elements of the same class")
- `.mix`: to represent a collection of element types from different classes

Common Attributes

The `.attrib` suffix is used to represent a group of PEs that themselves represent one or more complete attribute specifications. For instance, the frequently used `Common.attrib` is defined as follows:

```
<!ENTITY % Common.attrib
      "%Core.attrib;
       %I18n.attrib;
       %Events.attrib;
       %Common.extra.attrib;"
>
```

The first three entities correspond to the `coreattrs`, `i18n`, and `events` PEs of the HTML 4.0 DTD. `Common.extra.attrib` is the empty string that serves as a hook for possible customizations. Three more `.extra` PEs are declared for customizing element content models: `Inline.extra`, `Block.extra`, and `Misc.extra`. They appear as the last choice within some content models; to customize those models you redefine the `.extra` entity as a string that begins with the pipe character. (Compare our Listing 3-10 in the section on DTD reuse.) You have to do it in the right place, to make sure that your definition overrides the original one.

Inside the List Module

If we look inside the list module, we'll find all those PEs and more: everything down to each individual element's name, content model, and attribute declaration is parameterized. Listing 3-12 is the section of the list module that defines unordered list, `ul`. Brief explanations have been inserted in the listing, and more-detailed explanations follow it. We want to make it clear that, to actively use the modularization framework (and relatively few people will have a need to do that), you will have to spend more time with the W3C recommendation and examples.

Listing 3-12. An Excerpt from the List Module

```
<!-- ul: Unordered List (bullet styles) ...............-->
<!ENTITY % ul.element  "INCLUDE" >
<![%ul.element;[
<!ENTITY % ul.content  "( %li.qname; )+" >
<!ELEMENT %ul.qname;  %ul.content; >
<!-- both the qname and the content model of each element are PEs,
so they can be modified: qname, to modify the prefix; content model,
to extend the definition as needed -->
<!-- end of ul.element -->]]>

<!ENTITY % ul.attlist  "INCLUDE" >
<![%ul.attlist;[
<!ATTLIST %ul.qname;
       %Common.attrib;
>
<!-- end of ul.attlist -->]]>
```

```
<!-- li: List Item .....................................-->

<!ENTITY % li.element  "INCLUDE" >
<![%li.element;[
<!ENTITY % li.content
     "( #PCDATA | %Flow.mix; )*"
><!-- .mix PEs combine block and inline content models -->
<!ELEMENT %li.qname;  %li.content; >
<!-- end of li.element -->]]>

<!ENTITY % li.attlist  "INCLUDE" >
<![%li.attlist;[
<!ATTLIST %li.qname;
      %Common.attrib;
>
```

Notice how each element's name is a `.qname` PE and each element's content is a `.content` PE so that each element's declaration has the following form:

```
<!ELEMENT %li.qname;  %li.content; >
```

Because the element's name is a PE that starts with a `prefix` PE, the DTD is namespace ready. Because its content is a PE, it can be extended or overridden. To see how it all works together, let's take a look at XHTML Basic, an XHTML subset for mobile devices.

XHTML Basic DTD

The DTD consists of the driver file (xhtml-basic10.dtd), the content model file (xhtml-basic10-model-1.mod), and a number of MOD files for individual modules. The driver file defines "globals," such as the namespace prefix, and includes XHTML modules. The content model file overrides selective content models in the included modules. To customize, you include what you want and create your own content model file.

The entire modularization framework package of DTD, module, and entity-definition files can be downloaded as a zipped archive from the W3C Web site.

The Driver File

Because in DTDs the first definition overrides later ones, the content model file is the first to be included. It is followed by the xhtml-framework module that has to

be included in a DTD for any XHTML-modules language because the module contains PE definitions for the basic things such as data types, notations, and namespace-qualified names. Because of complex interdependencies that should not concern us here, the content model for the `pre` element must be redefined at this point. (The HTML `pre` element, as you recall, serves much the same function as the `CDATA` section of XML.)

The next three items include three out of four required modules (text, hypertext, and list) that must be included in any language that aspires to belong to the XHTML modularization family. The bulk of the rest is taken up by including the other modules, not required for all XHTML-based languages but required for XHTML Basic. The file concludes with the remaining required module, the structure module, which defines such elements as `html`, `head`, and `body`.

We show the driver file with almost all comments, a few obscure lines, and repetitive material removed, as indicated by our comments. The code that includes modules and our own comments are highlighted. Listing 3-13 shows the first part, up to the xhtml-text module, and Listing 3-14 shows the second part.

Listing 3-13. The First Part of the Driver File, xhtml-basic10.dtd

```
<!ENTITY % XHTML.version  "-//W3C//DTD XHTML Basic 1.0//EN" >
<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % XHTML.prefix  "" >
<!-- Reserved for use with the XLink namespace: -->
<!ENTITY % XLINK.xmlns "" >
<!ENTITY % XLINK.xmlns.attrib "" >
<!-- a few lines removed; the content model module is next,
      followed by the framework module -->
<!ENTITY % xhtml-model.mod
     PUBLIC "-//W3C//ENTITIES XHTML Basic 1.0 Document Model 1.0//EN"
            "xhtml-basic10-model-1.mod" >
<!ENTITY % xhtml-framework.mod
     PUBLIC "-//W3C//ENTITIES XHTML Modular Framework 1.0//EN"
            "xhtml-framework-1.mod" >
%xhtml-framework.mod;
<!ENTITY % pre.content
     "( #PCDATA
      | %InlStruct.class;
      %InlPhras.class;
      %Anchor.class;
      %Inline.extra; )*"
>
<!ENTITY % xhtml-text.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Text 1.0//EN"
            "xhtml-text-1.mod" >
%xhtml-text.mod;
```

This is followed by two more required modules: hypertext and list. They are "required" in the sense that they are included unconditionally and cannot be easily removed in languages that are derived from XHTML Basic. By contrast, the "optional" modules are included conditionally and can be turned off by simply redefining the `.module` PE. We show the first three out of eight optional modules to point out the difference between the tables and forms modules on the one hand and the rest of them, exemplified by the image module. If you look at the FPIs within their `.mod` PE definitions, you will see that the image module is adopted unchanged from the general XHTML framework but that the tables and forms modules have been hand crafted for XHTML Basic. It would be more in the spirit of modularization of XHTML to derive the Tables and Forms modules from Basic Tables and Basic Forms modules, as it is done in RELAX NG. (An XML Schema framework for modularization has not yet been implemented.)

Listing 3-14. The Second Part of the Driver File, xhtml-basic10.dtd

```
<!-- Image Module ...............................................-->
<!ENTITY % xhtml-image.module "INCLUDE" >
<![%xhtml-image.module;[
<!ENTITY % xhtml-image.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Images 1.0//EN"
            "xhtml-image-1.mod" >
%xhtml-image.mod;]]>
<!-- Tables Module ...............................................-->
<!ENTITY % xhtml-table.module "INCLUDE" >
<![%xhtml-table.module;[
<!ENTITY % xhtml-table.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Basic Tables 1.0//EN"
            "xhtml-basic-table-1.mod" >
%xhtml-table.mod;]]>
<!-- Forms Module ...............................................-->
<!ENTITY % xhtml-form.module "INCLUDE" >
<![%xhtml-form.module;[
<!ENTITY % xhtml-form.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Basic Forms 1.0//EN"
            "xhtml-basic-form-1.mod" >
%xhtml-form.mod;]]>
<!-- followed by the link, meta, base, param, and object modules,
      same as in XHTML -->
```

Finally, in the very end, the structure module is included, unconditionally:

```
<!ENTITY % xhtml-struct.mod
     PUBLIC "-//W3C//ELEMENTS XHTML Document Structure 1.0//EN"
            "xhtml-struct-1.mod" >
%xhtml-struct.mod;
<!-- end of XHTML Basic 1.0 DTD.............................................-->
```

The Content Model File

The content model file, xhtml-basic10-model-1.mod, consists of four sections. The brief first section defines the optional elements within the head element and the `misc.class` entity that is a placeholder for future extensions. Recall that all the `.qname` entities are declared in the qname module, which is included in the framework module:

```
<!ENTITY % HeadOpts.mix  "( %meta.qname; | %link.qname; | %object.qname; )*" >
<!ENTITY % Misc.class "" >
```

The bulk of the content model module is taken up by entities for inline and block elements. Listing 3-15 shows the inline elements section.

Listing 3-15. The Content Model File, xhtml-basic10-model-1.mod

```
<!ENTITY % InlStruct.class "%br.qname; | %span.qname;" >
<!ENTITY % InlPhras.class
     "| %em.qname; | %strong.qname; | %dfn.qname; | %code.qname;
      | %samp.qname; | %kbd.qname; | %var.qname; | %cite.qname;
      | %abbr.qname; | %acronym.qname; | %q.qname;" >
<!ENTITY % InlPres.class "" >
<!ENTITY % I18n.class "" >
<!ENTITY % Anchor.class "| %a.qname;" >
<!ENTITY % InlSpecial.class "| %img.qname; | %object.qname;" >
<!ENTITY % InlForm.class
     "| %input.qname; | %select.qname; | %textarea.qname;
      | %label.qname;"
>
<!ENTITY % Inline.extra "" >
<!ENTITY % Inline.class
     "%InlStruct.class;
      %InlPhras.class;
      %Anchor.class;
      %InlSpecial.class;
      %InlForm.class;
      %Inline.extra;"
>
<!ENTITY % InlNoAnchor.class
```

```
     "%InlStruct.class;
      %InlPhras.class;
      %InlSpecial.class;
      %InlForm.class;
      %Inline.extra;"
>
<!ENTITY % InlNoAnchor.mix
     "%InlNoAnchor.class;
      %Misc.class;"
>
<!ENTITY % Inline.mix
     "%Inline.class;
      %Misc.class;"
>
```

As we said, a complete active mastery of this material requires careful reading of the specification and examples. Within the constraints of this chapter, we can provide a couple of details that may help understanding.

Entities that get concatenated with `InlStruct` start with the pipe character, just as the `morecolors` PE in our Listing 3-10. When a hook is needed for possible future customizations, we declare an entity whose value is the empty string; this is the role of `Inline.extra` and `Misc.class`. In customizations, the values of those entities will also have to start with the pipe character. This is, in effect, how we express in our "modularization language" the difference between choice and sequence in content models: PEs for those elements that appear as part of a sequence start with the pipe, except the first one, chosen more or less arbitrarily.

The section of block elements follows the same structure. We will show a small portion of it:

```
<!ENTITY % Heading.class
     "%h1.qname; | %h2.qname; | %h3.qname;
      | %h4.qname; | %h5.qname; | %h6.qname;"
>
<!ENTITY % List.class  "%ul.qname; | %ol.qname; | %dl.qname;" >
<!ENTITY % Table.class "| %table.qname;" >
<!ENTITY % Form.class  "| %form.qname;" >
<!ENTITY % BlkStruct.class "%p.qname; | %div.qname;" >
```

As before, entities that get added to other entities start with the pipe character.

The final section declares “flow” entities, which are catch-all entities that include all content elements, both block and inline. Because tables are not appropriate for many small screens, two versions of flow are declared, with and without tables:

```
<!ENTITY % FlowNoTable.mix
     "%Heading.class;
      | %List.class;
      | %BlkStruct.class;
      %BlkPhras.class;
      %Form.class;
      %Block.extra;
      | %Inline.class;
      %Misc.class;"
>
<!ENTITY % Flow.mix
     "%Heading.class;
      | %List.class;
      | %Block.class;
      | %Inline.class;
      %Misc.class;"
>
<!-- end of xhtml-basic10-model-1.mod -->
```

Summary: Is This Worth Doing?

Modularization of XHTML is careful to note that “most users of XHTML are *not* expected to be DTD authors.” Indeed, developing a custom DTD within the XHTML modularization framework is an intricate and time-consuming affair. If you want to try your hand at it, study *XHTML Basic* or the RDDL DTD at `http://www.rddl.org`. Another example is the *MathML 2.0* recommendation that comes with a ready-made DTD for combined XHTML and MathML. Our own practice is that *if* you need an XHTML-based language *and* you need to validate it *and* none of the existing DTDs fits your needs, then use RELAX NG and save yourself a good deal of aggravation.

What's Wrong with the DTD?

Collecting the observations we have accumulated so far, we have this list of DTD grievances:

- They have non-XML syntax.
- Element content models and attribute lists can be modified only as PE values that are unstructured strings of characters.
- They are namespace unaware.
- They have a very primitive system of data types.

Both RELAX NG and XML Schema address all of these issues, in very different ways, as we will see in Chapter 8.

The DTD, the XML Document, and the Infoset

At this point, we have shown or mentioned everything that can be found in a DTD:

- element and attribute definitions
- general and parameter entity declarations and definitions
- notation declarations and definitions (mentioned only)
- general and parameter entity references

Here is a summary of everything that can found in an XML document:

- XML declaration
- document type declaration
- internal DTD
- elements and attributes
- comments, PIs
- general entity references and character references

- whitespace that is not part of any of the above (for example, between elements)
- irrelevant whitespace elsewhere (such as within the end tag of an element)
- CDATA sections

How much of this should be preserved in the document tree? Obviously, elements and attributes, but what about CDATA sections or comments? If a general entity is resolved before the tree is constructed, should its boundaries be preserved in the tree? The *Infoset* specification gives the authoritative answers to all these questions. As you recall, the *Infoset* recommendation is intended to harmonize tree models used in other recommendations, such as *DOM*, *XPath*, *XPointer*, and the forthcoming *XML Query Language*.

The Infoset Overview

Everybody is in agreement that infoset is abstract; the disagreement is over whether this is good or bad. In infoset's view, an XML document consists of abstract information items, of which there are eleven kinds. We get a complete list from the table of contents for Section 2 of the W3C recommendation:

- the document information item
- element information items
- attribute information items
- processing instruction information items
- unexpanded entity reference information items
- character information items
- comment information items
- the document type declaration information item
- unparsed entity information items
- notation information items
- namespace information items

We notice that unexpanded entity references are information items but the boundaries of expanded references are not. Appendix D of the recommendation lists “what is not in the Information Set” (but warns that the list does not claim to be exhaustive):

- the content models of elements, from ELEMENT declarations in the DTD
- the grouping and ordering of attribute declarations in ATTLIST declarations
- the document type name
- whitespace outside the document element
- whitespace immediately following the target name of a PI
- whether characters are represented by character references
- the difference between the two forms of an empty element: `<foo/>` and `<foo></foo>`
- whitespace within start tags (other than significant whitespace in attribute values) and end tags
- the difference between CR, CR-LF, and LF line termination
- the order of attributes within a start tag
- the order of declarations within the DTD
- the boundaries of conditional sections in the DTD
- the boundaries of parameter entities in the DTD
- comments in the DTD
- the location of declarations (whether in internal or external subset or parameter entities)
- any ignored declarations, including those within an IGNORE conditional section, as well as entity and attribute declarations ignored because previous declarations override them

- the kind of quotation marks (single or double) used to quote attribute values
- the boundaries of general parsed entities
- the boundaries of CDATA marked sections
- the default value of attributes declared in the DTD

Future specifications (such as *XQuery*) and future versions of existing specifications (*XPath* and *DOM* in particular) will be defined in terms of information items and comply with the infoset requirements.

Conclusion

In this chapter, we covered DTDs and validation, asking both the practical programmer's question (how do you validate?) and more-analytical questions (What is validation for? What else are DTDs good and not good for? How are DTDs extended and reused?). An important distinction that applies both to DTDs and other validation tools (XML Schema, RELAX NG) is between validating a document and adding information to a document as a side effect of validation. (This is usually described as "augmenting the infoset of the document.") DTD can augment the infoset in several ways: general entities, attribute defaults, and ID/IDREFs constraints.

We have not talked nearly enough (if at all) about the role of DTDs in human affairs rather than within programs. The most important consumers of DTDs may not be validating parsers but human readers, for DTDs frequently function as:

- a human-readable model of some subject domain
- a design document
- a piece of documentation
- a contract between a group of people or organizations to use the same document format

Many industries and fields of knowledge have created (or are in the process of creating) their DTDs. See Robin Cover's pages at `http://xml.coverpages.org` for a comprehensive list. Some of these have started the process of conversion to XML Schema. When RELAX NG becomes standardized and better known, we expect its use to spread as well.

Another important use of DTDs and validation has to do with security. As XML is increasingly used to exchange information between programs, including parameters for remote procedures, the issue of XML security is becoming increasingly important. Validation is an important component of XML security. (However, like all security precautions, it does incur a performance penalty.) We will come back to the issue of security and validation in the chapter on Web services and SOAP.

CHAPTER 4

XML Parsing

THE PURPOSE OF PARSING an XML document is to expose some interfaces to an application that needs to work with the document's data. The XML parser sits in the middle between an XML document and an application that uses it. This is shown in Figure 4-1, which is repeated from Figure 1-3 of Chapter 1.

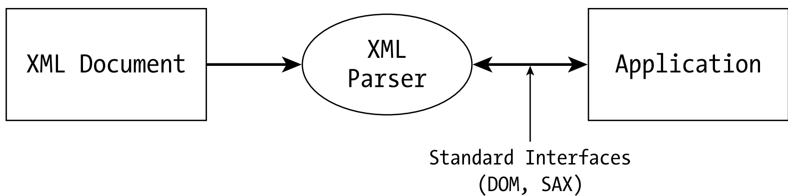

Figure 4-1. The document, the parser, and the application

As you know, the main reason for using XML is interoperability. There would be no interoperability if different parsers treated the same data differently or if they provided different interfaces. The behavior of an XML parser is heavily regulated on both ends of Figure 4-1. Its treatment of the XML document is precisely specified in *XML 1.0*, in the section on conformant processors. Its obligations towards the application are precisely specified in two APIs that form the subject matter of this chapter: the DOM (Document Object Model) and SAX (Simple API for XML).

The reasons there are two APIs have to do with space-time tradeoffs, which are so common in programming. DOM lays out the document in (memory) space as a tree data structure that is available to the application all at once. You can traverse and edit the tree, as long as there is enough memory to store it. (Several estimates suggest that a DOM tree needs at least three times as much memory as the XML document.) SAX requires a modest amount of memory (which is proportional to the depth of the document tree) because SAX lays out the document in time and as a sequence of events. It associates an event with each tag (opening or closing) and with each block of text. You just write the event handlers (also known as *callbacks*) and sit back to watch the document pass by. SAX is a very efficient and flexible tool, but it becomes awkward to use if the way you process a given element depends upon earlier or later elements in the document.

Note that either of the two packages could be implemented using the other one. You could generate SAX events by traversing a DOM data structure; later in the chapter, we will show how and explain why you would want to do such a thing. Conversely, you could build the DOM data structure by appropriate SAX event handlers. (Many XML parsers work this way, which is why, in Java code, you will see a DOM parser throw SAX exceptions.) Another way for DOM and SAX to work together is to apply a SAX parser to a very large document, filter the output to a smaller document, and convert that to DOM for further processing. DOM, SAX, and XSLT are well suited for processing chains or pipelines of that nature, and you will see many of them in the remainder of the book.

Although DOM is easy to understand, SAX may appear puzzling at first, even if you have done a good deal of GUI programming. (A recent discussion on the xml-dev list, `http://lists.xml.org/archives/xml-dev/200111/msg00180.html`, suggests that SAX is underappreciated and underused.) We will present it first. In outline, the chapter will proceed as follows:

- SAX overview
- obtaining a parser and setting its properties
- simple example
- more-advanced uses (SAX filters)
- JAXP transformer objects and SAX-to-document conversion
- SAX parsing for non-XML data
- DOM programming, including traversal interfaces
- DOM and SAX working together

Long before we get to the last item of this outline, we want to emphasize that a sequence of SAX events is just another representation of XML data, equivalent to text with markup or a DOM tree. All three are standard representations of XML. In this chapter, we will see a general-purpose tool for converting among the three representations.

Basic SAX Programming

Unlike most other languages and APIs used in working with XML, SAX does not come from W3C. W3C standardized DOM very soon after releasing *XML 1.0*

because a need for standard parser interfaces was clear. As people started using it, some drawbacks of the DOM approach became obvious: it takes a good deal of time and a lot of memory to construct a representation for a very large document; until it is constructed, you cannot do anything with it; and, if you want to filter the document's data and output only a small part of it, you have to parse the entire document and construct its entire DOM tree before you can start running your filter.

SAX came about in response to these and similar criticisms that were frequently voiced on the xml-dev list. The person who actually got it done was David Megginson (`http://www.megginson.com`). When SAX2 (with support for namespaces) was released, the project moved to SourceForge (`http://sax.sourceforge.net`). SAX is even less of a formal standard than W3C recommendations—it is not backed by any consortia—but it is very widely accepted, both by individual developers and by the likes of Sun and IBM, and so its stability is ensured, at least on the Java platform.

> **NOTE** *The .NET platform does not provide a SAX implementation. Instead, it implements a "pull" interface that allows the programmer to read in, or skip, the next element. The pull model has some advantages over SAX and has been implemented in Java also. (See* `http://www.kvmworld.com/articles/techtalk/kxml`*.)*

SAX, Java, and Other Languages

The FAQ at `sax.sourceforge.net` contains this exchange:

> *Where's the formal language-independent SAX2 Specification?*
>
> *There isn't any, and probably there won't ever be one. SAX2 in Java is defined by its interfaces and by the base of running code—it's more like English Common Law than the heavily codified Civil Code of ISO or W3C specifications. Outside of Java, SAX is whatever programmers in that language decide it should be.*

Despite this warning, there are a number of SAX implementations in languages other than Java, including Microsoft's MSXML 3.0 and later. (See the FAQ for links.) However, in describing SAX functionality, we will have to make use of Java interfaces, even if the implementation is in Visual Basic. (There is an interface definition in Microsoft IDL, but it is specific to COM/ActiveX.) We will present Microsoft's SAX implementation as used in VB 6 later in the chapter, in an application that uses DOM and SAX together.

The SAX Trio: Parser, Input Source, and Content Handler

A SAX application uses at least three objects that work closely with each other: a parser, a source of XML input, and a "handler" that processes the input in application-specific ways. The application creates a parser instance and provides it with two pieces of information: where the XML content is coming from (the input source) and who is going to process it (content handler). This can be done in a couple of different ways, depending on the framework in which the parser is instantiated. Suppose the names of the variables are `parser`, `input`, and `handler`, respectively. In the original SAX2 framework, the application would have these two lines of code:

```
parser.setContentHandler(handler);  // register the handler with the parser
parser.parse(input);
```

In Sun's JAXP framework (JAVA API for XML Processing), the application would have this line:

```
parser.parse(input, handler);
```

Whatever the setup code, after the `parse()` method is called, control is passed to the parser. The parser goes through the input data and passes pieces of information to the handler (the start tag name, attributes, text content, and so on) as they are encountered. How do the parser and the content handler communicate? The answer is that the content handler defines certain agreed-upon methods, and the parser calls them. The contract between the parser and the handler (enforced by SAX interfaces) is that if the parser, for instance, discovers the end tag of an element, it will call the handler's `endElement()` method with the tag name of the element as argument.

This is usually described in the terminology of events and callbacks. A *callback*, in case you are unfamiliar with the term, is a function that you implement but don't call: it gets called by the system or framework in response to some event. Callbacks are common in GUI programming: the user clicks on a button and in response some `onclick()` function, written specifically for this occasion, gets called. In the case of the parser and the handler, the events received by the handler are not GUI events but the important events in parsing an XML document: the document has started, an element has started, an element has ended, the character content of an element has been found, and so on. SAX provides standard names for callback functions that are triggered by these events. Writing a SAX application usually consists of implementing those callbacks.

Callback Illustration

Consider the document of Listing 1-5 (back in Chapter 1):

```
<exchange><q>What's up?</q><a>Nothing much.</a></exchange>
```

Assuming, as before, that the name of the content handler variable is `handler`, the following calls will be made by the parser, in the order shown in Listing 4-1.

Listing 4-1. Sequence of SAX Callbacks

```
handler.startDocument()
handler.startElement("exchange")
handler.startElement("q")
handler.characters("What's up?")
handler.endElement("q")
handler.startElement("a")
handler.characters("Nothing much.")
handler.endElement("a")
handler.endElement("exchange")
handler.endDocument()
```

This is a simplified and slightly inaccurate picture: the reality is a bit more complex because of namespaces, attributes, and the limited size of character buffer. Specifically:

- If the parser is configured to be namespace aware, then it calls `startElement()` with three String arguments: local name, namespace URI, and qName (qualified name).
- Whether the parser is namespace aware or not, `startElement()` also has an Attributes argument. If there are no attributes, the argument is null.
- There is no commitment that the text content of an element will be delivered in a single `characters()` call. If the text content of an element is large, it may be spread over several calls on the `characters()` method.

We will present complete APIs shortly.

Other Handlers, InputSource, and the Locator

In addition to a content handler, the application can register several other handlers with the parser. They are defined in `org.xml.sax` and `org.xml.sax.ext`.

- DTDHandler is of limited use because it reports only notations and non-XML entities. Because we never use either of those, we never use DTDHandler.
- DeclHandler exposes individual declarations in the DTD. The methods are `internalEntityDecl()`, `externalEntityDecl()`, `elementDecl()`, and `attributeDecl()`.
- LexicalHandler provides access to lexical material other than element content and attribute values. Its methods are `comment()`, `start/endDTD()`, `start/endCDATA()`, and `start/endEntity()`.
- ErrorHandler allows the programmer to write custom handlers for both errors and warnings generated by the parser.
- EntityResolver is useful when you know that your document contains external entities and you want to make sure that they are processed correctly, rather than relying on the parser's default behavior.

EntityResolver

EntityResolvers must implement exactly one method:

```
public InputSource
    resolveEntity(java.lang.String publicId,  java.lang.String systemId)
    throws SAXException, java.io.IOException
```

The method, and the entire EntityResolver object, is useful in several typical situations:

- You use a PUBLIC identifier, perhaps resolved by accessing a local registry.
- You use a URI that is not a URL and requires special treatment.
- Your entity content is formatted in a specific way that requires preprocessing before it is submitted to the standard parser.

Whatever the reason, the object returned by `resolveEntity()` is an InputSource object, something the parser understands and knows how to parse. InputSource is a class (not an interface), and InputSource objects can be delivered to the parser in one of two ways: as the output of the EntityResolver that is registered with the parser, or directly as an argument to the `parse()` method.

InputSource and Other Sources of Input

In SAX2, the interface that declares the methods of a parser object, including the `parse()` method, is called XMLReader. Two versions of `parse()` are declared:

```
void parse(java.lang.String systemId)
void parse(InputSource input)
```

In other words, you can give the parser a URL (which must be fully resolved), or you can give it an InputSource object. An InputSource object, in turn, can be created from a URL, a local file name, or a stream (text or binary). So, if your source is specified by a URL, you can give it to the parser in two equivalent ways:

```
parse(myUrl);  // myUrl is a String
parse(new InputSource(myUrl));
```

If your input comes from a stream, whether character or binary, then you don't have a choice: you have to wrap the stream into an InputSource object. If your input comes from a byte stream, the code may look like this:

```
InputSource input=new InputSource(myStreamSource);
input.setEncoding(myEncoding); // myEncoding is a String such as "utf-8"
input.setSystemID(baseURI); // baseURI is for resolving relative URLs within input
parse(input);
```

If your input comes from a (fully qualified) URL and you have a relative URL within it (for example, `<!DOCTYPE contacts SYSTEM "contacts.dtd">`), then your relative URLs are resolved relative to the absolute URL of the document. If your input comes from a stream of bytes or characters, then there is a problem: there is no way to determine from the input how relative URIs are to be resolved, unless you provide a base URI as a property of the InputSource object.

> **NOTE** *The recent XML Base recommendation (`http://www.w3.org/TR/xmlbase`) defines an `xml:base` attribute that can be used within a document to specify its base URI for resolving relative URIs. Its use is not yet common and cannot be relied on.*

Input from a File

If your input comes from a local file, you have to convert the file name to a URL. In Java, this is easily done using the File and URL classes: a File object knows how to convert itself to a URL object, and a URL object knows how to convert itself to its string name:

```
parse((new File(fileName)).toURL().toString());
```

where `fileName` is an absolute filename such as C:\data\test.xml.

Creating a Parser

As we said, a SAX application consists of two parts. In the first part, we set things up:

1. Create a parser instance.
2. Register a content handler with it and possibly other handlers.
3. Obtain an input source.

This part ends when we say `parser.parse()` (and catch the exceptions). The second part defines callbacks (that is, the content handler's methods). They are contained in a class that implements the ContentHandler interface. The SAX distribution contains a package of "helper" classes, including DefaultHandler, a class that provides a default, do-nothing implementation for all the methods of ContentHandler. Usually, ContentHandler is not implemented directly, but rather the DefaultHandler is extended to provide definitions of those callbacks that our application needs to use. You will see how this works in our example.

In this section, we concentrate on the setup part. Java has two main approaches: SAX2-specified interfaces and Sun's JAXP (Java APIs for XML Processing).

The SAX2 Way

If you don't mind hardwiring a specific parser into your application, you can simply say

```
XMLReader myReader =
    new org.apache.xerces.parsers.SAXParser(); // or another parser of your choice
```

A better way is to use the XMLReaderFactory class that is included in SAX2 distribution. The class has a public static method for creating parser instances:

```
String parserClass="org.apache.xerces.parsers.SAXParser";

                                // or another parser of your choice
try {
   XMLReader myReader =
      XMLReaderFactory.createXMLReader(parserClass);
 } catch (SAXException e) {
   System.err.println(e.getMessage());
 }
```

The method can also be called with no argument, in which case the factory class will look for the value of the System property `org.xml.sax.driver`. Your code is now portable between all installations or contexts in which this property is defined.

The JAXP Way

JAXP is a collection of lightweight, abstract wrapper classes that make it easy to instantiate and configure a parser. JAXP also contains abstract classes for creating very efficient XSLT processors, each implementing a specific XSLT stylesheet compiled into binary code. We will see XSLT processors later in the chapter; for now, we concentrate on parsing.

JAXP classes for creating parsers are as follows:

- DocumentBuilder (DOM parser)
- DocumentBuilderFactory
- SAXParser
- SAXParserFactory

This is how you create a parser using JAXP:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
// set properties as needed
factory.setValidating(true); // default is false, non-validating
factory.setNamespaceAware(true);  // default is false
SAXParser saxParser = factory.newSAXParser();
```

The JAXP SAXParser class is a wrapper around a SAX2 XMLReader object. You can extract the XMLReader object and use it as described in the preceding section:

```
XMLReader reader=saxParser.getXMLReader();
```

Alternatively, you can use the JAXP SAXParser directly. It has its own `parse()` method that provides several convenient features. For instance, you can use a File object directly:

```
// filename is a String that is a file name; handler is a ContentHandler
saxParser.parse( new File(filename), handler);
```

Which parser gets called? Remember that SAX and DOM are just interfaces and default “do-nothing” implementations for some of them; to get anything done (like parse a document), the interfaces have to be implemented or default implementations extended. JAXP itself does not do that: it consists of four abstract classes that provide factory methods for obtaining instances of a parser and an XML data source. Specific implementations have to extend the abstract classes. As part of the JAXP distribution (but not part of JAXP itself), Sun provides a reference implementation: the Crimson parser in `org.apache.crimson.jaxp.SAXParserImpl`. Crimson is a relatively small parser that does not have the same functionality as Xerces: it supports namespaces but not XML Schema validation or Xerces’ serialization and XPath features. If you want to use Xerces, you have to set the System property that creates a SAXParserFactory:

```
javax.xml.parsers.SAXParserFactory = org.apache.xerces.parsers.SAXParser
```

The actual code that uses JAXP classes is not affected in any way.

The ContentHandler Interface

So far, we have been talking mostly about technicalities and preparatory actions. The substance of the application is the ContentHandler callbacks. Here are the most important declarations:

```
// Receive notification of the beginning and end of a document.
void startDocument();
void endDocument();
// Receive notification of the beginning and end of an element.
void startElement(
    java.lang.String namespaceURI,
    java.lang.String localName,
    java.lang.String qName,
    Attributes atts
);
void endElement(
    java.lang.String namespaceURI,
    java.lang.String localName,
    java.lang.String qName
);
```

We will briefly comment on these declarations before going through an example.

`startDocument()–endDocument()`

`startDocument()` is called before any other callbacks, across all handlers, including the DTDHandler. The last method to be called is `endDocument()`; it gets called even if the `ErrorHandler` has been notified of an unrecoverable error.

`startElement()–endElement()` *and Namespaces*

These are workhorse methods that usually do most of the work. The values of their arguments depend on how the parser is configured. Within SAX2, the configuration of the parser is determined by two features, each of which can be set to True or False. One is the namespaces feature, whose name (which looks like a URL but isn't) is `http://xml.org/sax/features/namespaces`; it controls whether the parser is aware of namespaces at all. If it is set to False, the xmlns declarations are reported as attributes and element names are reported as qNames, with prefixes included. If the namespaces feature is set to True, then there is the question

of whether to report the xmlns declarations as attributes. This is controlled by the namespace-prefixes feature, whose name is `http://xml.org/sax/features/namespace-prefixes`. The four combinations of feature values are summarized in Table 4-1, which is adapted from `http://sax.sourceforge.net/?selected=namespaces`. You will be able to experiment with different options in the example application of the next section.

Table 4-1. SAX Parser Configurations

NAMESPACES	NAMESPACE-PREFIXES	NAMESPACE NAMES START/END	QNAMES XMLNS	ATTRIBUTES
True	False	Yes	Unknown	No
True	True	Yes	Yes	Yes
False	False	Illegal mode		
False	True	Unknown	Yes	Yes

In practice, if your parser is namespace aware, you usually don't want namespace declarations reported as attributes. This leaves you with just two possible states, corresponding to the first and last rows of the table. If you work with the JAXP SAXParser rather than SAX2 XMLReader, then you toggle between these two states using the `setNamespaceAware()` method of SAXParserFactory that sets the two features of its XMLReader to opposite values (True-False or False-True). If the parser is not namespace aware, then `namespaceURI` and `localName` are empty strings, and xmlns declarations are reported as attributes. If the parser *is* namespace aware, then all three strings are delivered, but xmlns declarations are not reported. In this case, `namespaceURI` will be empty only if no namespaces are declared. Local and qualified names will be the same in no namespace and in default namespaces.

`startElement()` *and Attributes*

In addition to element names and namespaces, the `startElement()` method receives the element's attributes, if any. They are delivered in an object that implements the Attributes interface. SAX2 distribution contains a class that implements the interface, `AttributesImpl`.

Even though attributes are unordered, the Attributes interface provides access to them by a 0-based integer index. The index is assigned arbitrarily by the implementation. Given an index, you can find out everything there is to know about the corresponding attribute: its local name, qName, namespace URI, value, and data type. A common way of processing attributes is a For loop:

```
for(int i=0;i<attr.getLength();i++){
   String attrQName=attr.getQName(i);
   String attrLocalName=attr.getLocalName(i);
   // same for uri, value and data type if needed; process as needed
}
```

You can also find out the value and the data type by the qName or by a combination of the local name and namespace URI.

There is more to the ContentHandler interface, but we have covered enough to write a simple but meaningful example, a program that will compile statistics on how many times every element name, attribute name, and namespace URI appears in the document.

An Example

The entry page to the program (see Figure 4-2) offers a text area to enter an XML document and a selection element to set namespace awareness to True or False. You can also specify a document by a URL.

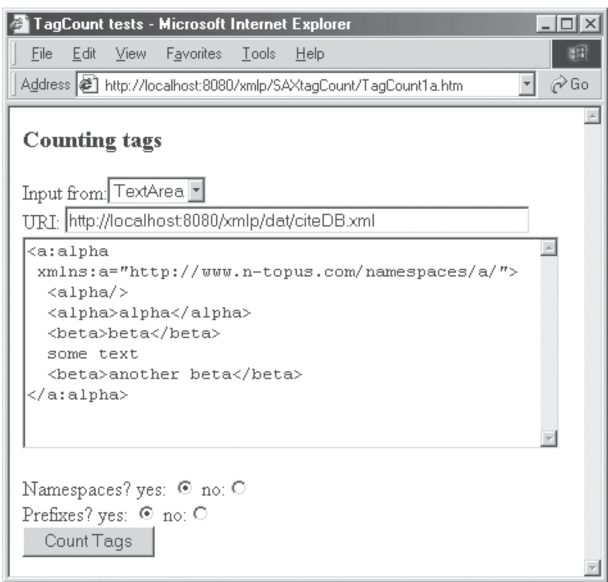

Figure 4-2. CountTags, entry page

With the document as shown in the text area and the parser set to be namespace aware, the output appearing in the text area is shown in Listing 4-2, with some whitespace rearranged.

Listing 4-2. Namespace-Aware Parsing Output

```
<ht:Hashtable xmlns:ht="http://www.n-topus.com/namespaces/ht">
<ht:item key="uri-http://www.n-topus.com/namespaces/a" >1</ht:item>
<ht:item key="local-beta" >2</ht:item>
<ht:item key="qName-a:alpha" >1</ht:item>
<ht:item key="qName-beta" >2</ht:item>
<ht:item key="uri-" >4</ht:item>
<ht:item key="qName-alpha" >2</ht:item>
<ht:item key="local-alpha" >3</ht:item>
</ht:Hashtable>
```

It shows there is one element in the `http://www.n-topus.com/namespaces/a` namespace and four elements in no namespace. (This is the meaning of "4" occurrences of empty `uri`.) These are elements whose `qname` is identical to their local name. There are two such `beta` elements and two (out of three) `alpha` elements; the remaining `alpha` element has the local name `alpha` but the `qname` `a:alpha`. Note that `alpha` and `a:alpha` are counted as having the same local name.

If we change the setting to non-namespace aware, we will get different results, as shown in Listing 4-3.

Listing 4-3. Output from Non-Namespace-Aware Parsing

```
<ht:Hashtable xmlns:ht="http://www.n-topus.com/namespaces/ht">
<ht:item key="local-" >5</ht:item>
<ht:item key="attQName-xmlns:a" >1</ht:item>
<ht:item key="qName-a:alpha" >1</ht:item>
<ht:item key="qName-beta" >2</ht:item>
<ht:item key="uri-" >5</ht:item>
<ht:item key="qName-alpha" >2</ht:item>
</ht:Hashtable>
```

This time, all URIs and local names are empty, and `xmlns:a` is counted as an attribute name. You can experiment by editing the document in the text area and viewing the output under either setting.

The Code

The HTML page shown in Figure 4-2 displays a form whose action attribute is TagCount1a.jsp. The idea of the program is quite simple: each name and namespace URI is used as a hashtable key whose value is the count of this name's occurrences in the document. In processing the start tag of each element, the appropriate key is incremented. In outline, the code proceeds as follows:

1. Import libraries.

2. Define a content handler.

3. Instantiate input source and parser. Configure parser. Parse.

4. Output results (the contents of the hashtable).

We will show the code in two installments.

Import Libraries, and Define a Content Handler

Our content handler defines just one method, `startElement()`. The method, as you know, has four arguments: strings for namespace URI, local name, qName, and an Attributes object. For each of the three strings, we want to perform the same computation:

- Use the string as a key to retrieve the corresponding value from the hashtable.

- If the value is Null, set it to 1; otherwise, increment the value by 1.

One way to express it in Java is by a function that takes a hashtable and a string key as arguments. Because Java hashtables store data as generic objects, you have to convert each int into an Integer object before putting it into a hashtable, and you have to cast generic Object objects to Integer objects when taking it out. (See Listing 4-4.)

Listing 4-4. Converting Simple Type to Object Type in Java

```
void integerIncr(Hashtable ht,String key){
  Integer N=(Integer) ht.get(key);
  if (null==N) ht.put(key,new Integer(1));
  else ht.put(key, new Integer(1+N.intValue()));
 }
```

More compactly (and perhaps too densely), the same computation can be expressed as:

```
Integer uriInt=(Integer) ht.get(uri);
ht.put(uri, new Integer((null==uriInt)?1:1+uriInt.intValue()));
```

and similarly for the other two strings. The code shown in Listing 4-5 uses Listing 4-4.

Listing 4-5. Content Handler Definition

```
<%@ page errorPage="../error.jsp"
import="javax.xml.parsers.SAXParserFactory,
             javax.xml.parsers.SAXParser,
             org.xml.sax.helpers.DefaultHandler,
             org.xml.sax.Attributes,org.xml.sax.InputSource,
             org.xml.sax.helpers.DefaultHandler,  // helper class
              java.util.Hashtable,java.util.Enumeration"
%> <%! // declare integerIncr()
  // Listing 4-4 goes here
%><%!
 class KeyCounter extends DefaultHandler { // this is our ContentHandler
   Hashtable ht;
   public KeyCounter(Hashtable ht){this.ht=ht;} // constructor accepts Hashtable
   public Hashtable getHashtable(){return ht;}
   public void startElement( // define the callback
       String uri, String local, String qName, Attributes attr ){
  // add an appropriate prefix to each String
     uri="uri-"+uri; local="local-"+local;qName="qName-"+qName;
  // for each String argument,  increment or set to 1 its value in Hashtable
     integerIncr(ht, uri);  integerIncr(ht, local); integerIncr(ht, qName);
  // do the same with attributes, if any
     for(int i=0;i<attr.getLength();i++)
       integerIncr(ht,"attQName-"+attr.getQName(i));
   }
}
```

Parsing with JAXP

The part of the code shown in Listing 4-6 sets up the parser and calls its `parse()` method.

Listing 4-6. Set up Parser and Parse Using JAXP Methods

```
// extract URI and text area content from Request
 String uriStr=request.getParameter("uri");
 String docStr=request.getParameter("doc");
 String useTextArea=request.getParameter("inputSource");
// set up InputSource
 InputSource iS;
 if(!"yes".equals(useTextArea))
   iS=new InputSource(uriStr); // from URI
   else iS=new InputSource(new java.io.StringReader(docStr)); // from text area
```

```
// create a ContentHandler with a HashTable
 Hashtable ht=new Hashtable();  // to hold tag name counts
 KeyCounter keyCounter=new KeyCounter(ht); // this is our ContentHandler
// create and configure the parser
 SAXParserFactory factory=SAXParserFactory.newInstance();
 if("yes".equals(request.getParameter("namespaces")))
   factory.setNamespaceAware(true);
SAXParser parser=factory.newSAXParser();
// all set; parse! (callbacks are set in motion; data is stored in Hashtable
 parser.parse(iS,keyCounter);
```

Output Using JSP Code

In this first version, we will output the contents of the hashtable using Java and JSP. We extract the keys of the hashtable into an Enumeration object that has two methods: `nextElement()` and `hasMoreElements()`. They are usually deployed together in a loop, as in Listing 4-7. When you reach the end of the enumeration, `hasMoreElements()` returns Null.

In the body of the loop, we get a value for each key and output an `ht:item` element with two attributes: `key` and `val`. The values of the XML attributes are the values of our Java variables, so we use the `<%= %>` JSP construct:

```
<ht:item key="<%= key %>" ><%= val %></ht:item>
```

All together, the output code is shown in Listing 4-7.

Listing 4-7. Code to Display Parser Output in Text Area

```
// output the contents of the Hashtable
%><textarea rows="15" cols="65">
<ht:Hashtable xmlns:ht="http://www.n-topus.com/namespaces/ht">
<%
 for(Enumeration e=ht.keys();e.hasMoreElements() ;){
   Object key=e.nextElement();
   Object val=ht.get(key);
%><ht:item key="<%= key %>" ><%= val %></ht:item>
<% } %>
</ht:Hashtable>
```

The entire program of TagCount1a.jsp consists of Listing 4-5, 4-6, and 4-7. In version 1b, we will replace Listing 4-6 to show parsing without JAXP. In version 2, we will replace Listing 4-7 with code that outputs the hashtable data by converting it to XML. This will demonstrate a general technique for converting non-XML data to XML using a SAX parser.

Creating a Parser Without Using JAXP

The next version uses pure SAX2 (rather than JAXP) to produce the same result. Because we now have more fine-grained control over namespaces and xmlns declarations, we need an input element for each. Our user interface is now a two-frame arrangement so the user can see both the XML text and its tag count at the same time (TagCountFrames.htm and TagCount1b.htm). (See Figure 4-3.)

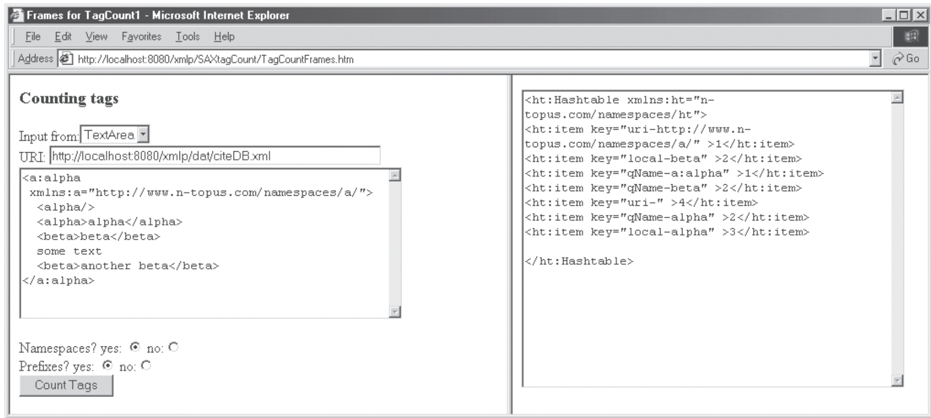

Figure 4-3. TagCount with frames

In the code, we make the following changes (TagCount1b.jsp).

1. Within imports, we import one more helper class:

```
import org.xml.sax.helpers.XMLReaderFactory;
```

2. We declare a String variable to hold the parser class name, get its value from the input form and use it to create a parser:

```
String parserClassName=request.getParameter("parserClass");
if (null==parserClassName)
  parserClassName="org.apache.xerces.parsers.SAXParser"; // Xerces by
default
XMLReader xmlReader =
     XMLReaderFactory.createXMLReader(parserClassName);
```

3. Because we can control both the namespace and namespace-prefixes features, we provide code for both:

```
  boolean ns="yes".equals(request.getParameter("namespaces"));
  xmlReader.setFeature("http://xml.org/sax/features/namespaces", ns);
  boolean pref="yes".equals(request.getParameter("prefixes"));
  xmlReader.setFeature("http://xml.org/sax/features/namespace-
prefixes",pref);
```

4. Finally, we set the ContentHandler to be our keyCounter class, and parse:

```
xmlReader.setContentHandler(keyCounter);
xmlReader.parse(iS);
```

The ContentHandler class of Listing 4-5 and the output code in Listing 4-7 remain unchanged. In the next version, Listing 4-7 will be replaced with code that converts a hashtable into XML data and serializes that data to output. Converting non-XML data (such as hashtable) to XML is an important and fairly advanced topic that we will cover later in the book. For now, it's time to take a break from hashtables and introduce SAX filters.

SAX Filters

A SAX filter is a SAX parser (XMLReader) that owns or otherwise controls another such parser. That second parser does the actual parsing and feeds SAX events to the filter. The filter can either suppress an event or forward it on to its content handler. Extending the metaphor of input sources and event streams, we can say that the second parser is positioned *upstream* from the filter.

In order for the filter to receive events from its upstream parser, the filter must implement the callbacks. In other words, a filter is both a parser and a content handler that is *the* content handler for its upstream parser. As a content handler, it inspects the events coming from the upstream parser. As a parser, it sends the events that pass its inspection to its content handler. (See Figure 4-4.)

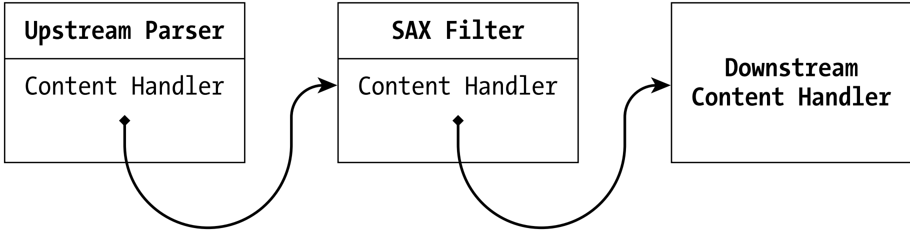

Figure 4-4. SAX filter, upstream parser, and content handlers

Suppose that we are writing a filter, the name of its upstream parser is `upParser`, the name of the handler is `handler`, and our filter has a `flipCoin()` procedure that returns either 0 or 1. It can then have code like this:

```
 public void startElement(
// as ContentHandler, we process callbacks from upstream parser
     String uri, String local, String qName, Attributes attr ){
     if(flipCoin()==1) // success
// as parser, we have registered a handler and can call its methods
      handler.startElement(uri, local, qName, attr);
    // else do nothing
}
```

This would make it difficult to filter out the corresponding endElement events because, by the time the handler receives such an event, it will have no idea whether or not the corresponding startElement passed or failed, but this is not a realistic example: most filters do a more careful inspection of the data coming from upstream.

SAX Support for Filter Writing

One of the Java SAX interfaces is XMLFilter, which extends XMLReader. In other words, a class that implements an XMLFilter has to implement all of XMLReader, including `parse()`. In addition, XMLFilter must have a "parent" that is also an XMLReader. This is the upstream parser from which the filter receives events. The parent may or may not be literally a parent class, in the object-oriented programming sense. If it is, the filter can refer to its upstream parser as *super*. (In Java, as in C++, an object refers to its parent object as *super*, and to itself as *this*.) If it is not, the `getParent()` and `setParent()` methods that every filter must implement provide access to the upstream parser.

The most common way to implement a filter class is to extend the XMLFilterImpl class provided in `org.xml.sax.helpers`. This class implements both XMLReader and ContentHandler. The default implementation simply asks the upstream parser to parse, and forwards the events from the upstream parser to its own content handler. In skeletal form, the XMLFilter of "SAXversepicker/pickverses.jsp," our next example, looks like this:

```
// as a parser, we parse
public void parse(InputSource input) {
  super.parse(input); // but we simply ask "parent" to do so
}
// as the parent's content handler, we implement callbacks
```

```
public void startElement(
  java.lang.String uri, java.lang.String localName,
  java.lang.String qName, Attributes atts
)
{
     // but we simply pass them on to our own content handler
     // which we have by virtue of being a parser
  this.getContentHandler.startElement(uri,localname,qName,atts);
}
```

In classes that extend XMLFilterImpl, such as our examples in the rest of this section, this code works behind the scenes, in inherited methods. Applications override selected callbacks to perform the filtering they need.

We will work through two examples. One will inspect text content and will pass it on for further processing only if it contains some keywords. The other will inspect the value of the first child element and will pass the entire subtree on if that value satisfies a certain condition (say, if it is equal to a given string). In both cases, we'll use the King James Bible as data.

Verse Picker

In this application, we will output all verses (v elements) that contain a given string. The URL of the data and the string to match can be entered from an HTML form or as a query string following the URL. For example, this URL (broken into two lines)

```
http://localhost:8080/xmlp/SAXversePicker/pickverses.jsp?
    verseMatch=Bethlehem&xmlUri=/dat/jb/ot.xml
```

outputs all the verses in ot.xml that contain the word *Bethlehem*. The verses are output as a flat list of v elements, all of them children of the root verses element. The root element has an attribute that shows the match string:

```
<?xml version="1.0" encoding="UTF-8"?>
<verses stringMatch="Bethlehem">
<v>And Rachel died, and was buried in the way to Ephrath, which is Bethlehem.
</v>
<!-- many more verses -->
</verses>
```

The most important work in this program is done by the `characters()` callback. We will go over its code before presenting the entire program in a systematic way.

The `characters()` *Callback*

Text content is delivered from the parser to the handler via calls on `characters()`. They take three arguments: a buffer (which is an array of characters), the starting position of the current subarray to be processed, and the length of that subarray. There is no guarantee that the entire content of an element will be delivered in a single call, so we usually have to set up a buffer on the receiving side, as part of the handler, and accumulate text content in it until `endElement()` is called.

To skip text content of elements that are not `v` elements or descendants of them, we have to set up a Boolean variable that will indicate the state of the parser: inside a `v` element or outside one. The state will be set by `startElement()` and `endElement()` callbacks. This is a common feature of SAX programming: to maintain contextual information (where in the tree are we?), you have to set up a state machine in the handler. In this case, with only two states to worry about, a single Boolean variable is adequate. For three or four states, you can have two Booleans or an enumerated state type. (We will have an example of each later in the book.) If you discover that you have to maintain very complex state conditions, you probably need a DOM tree. As an intermediate solution, you may consider a stack, which is less structured than a tree but more structured than a Boolean.

Outline

In outline form, pickverses.jsp proceeds as follows:

1. Import libraries.
2. Define the SAX filter class, VersePicker.
3. Initialize parameters and process request.
4. Define the content handler for VersePicker.
5. Create a filter object and do the parsing and filtering.

We will present the first three sections, which have to do with filtering, in complete detail. The last two sections perform fairly mundane tasks that can be

automated. We will briefly outline the handmade solution before presenting the automated one, in a separate section on JAXP transformers.

Import Libraries

Imports come from the following sources:

- standard Java distribution (IO classes)
- javax.xml.parsers (because we are going to use JAXP)
- org.xml.sax (most of them)
- org.xml.sax.helpers (AttributesImpl and XMLFilterImpl)

To use a transformer for parsing and output, we will also need (in the next section) a number of classes from the `javax.xml.transform` library. The complete set of imported libraries is shown in Listing 4-8.

Listing 4-8. VersePicker Inputs

```
<%@ page errorPage="error.jsp" import=
          "java.io.IOException, java.io.InputStream, java.io.StringReader,
             javax.xml.transform.TransformerFactory,
            javax.xml.transform.OutputKeys,
            javax.xml.transform.Transformer,
            javax.xml.transform.sax.SAXTransformerFactory,
            javax.xml.transform.sax.TransformerHandler,
            javax.xml.transform.sax.SAXSource,
            javax.xml.transform.stream.StreamResult,
            javax.xml.transform.stream.StreamSource,
            javax.xml.parsers.SAXParserFactory,
            org.xml.sax.Attributes,
            org.xml.sax.SAXException,
            org.xml.sax.XMLReader,
            org.xml.sax.EntityResolver,
            org.xml.sax.ContentHandler,
            org.xml.sax.ErrorHandler,
            org.xml.sax.InputSource,
            org.xml.sax.helpers.AttributesImpl,
           org.xml.sax.helpers.XMLFilterImpl"   %>
```

Define VersePicker Class

Because we have already discussed parts of the code, including the `characters()` method, we will present the class in total, with comments:

```
<%!
class VersePicker extends XMLFilterImpl  {
  boolean inVerse=false; // true while we're in a "verse" element
  StringBuffer verseBuff=new StringBuffer(); // to accumulate character content
  String stringMatch=""; // initialized at construction
  AttributesImpl atts;

  public VersePicker(String match){
    super(SAXParserFactory.newInstance().newSAXParser().getXMLReader());
      //  the XMLReader is the upstream parser (the parent)
    stringMatch=match;
    inVerse=false;
    atts=new AttributesImpl();
  // prepare Attributes object for root element; the arguments to addAttribute
  // are: namespace URI, local name, qName, data type, value
    atts.addAttribute("","stringMatch","stringMatch","CDATA",stringMatch);
  }
  public void startDocument()throws SAXException{
  // Output start tag of root element, with stringMatch attribute.
  // We say "this.getContentHandler" instead of simpy "getContentHandler"
  // to emphasize that we're using the content handler of this filter object
    this.getContentHandler().startDocument();
    this.getContentHandler().startElement("","verses","verses",atts);
    atts.clear();  // clear Attributes object for remaining (v) elements
  }
  public void endDocument()throws SAXException{
  // output end tag of root element
    this.getContentHandler().endElement("","verses","verses");
    this.getContentHandler().endDocument();
  }

  public void startElement(String nsURI,String localName,String qName,
                     Attributes atts) throws SAXException{
  // check qName, set the boolean flag accordingly
     if("v".equals(qName)) inVerse=true; else inVerse=false;
  }
```

```
  public void characters(char[]ch,int from, int len)throws SAXException{
// if in verse, append content to verse buffer
    if(inVerse) verseBuff.append(ch,from,len);
  }

  public void endElement(String nsURI,String localName,String qName)
        throws SAXException{
    if(inVerse && 0 <= verseBuff.toString().indexOf(stringMatch)){
  // if we are in a verse, and its text contains the match string,
  // send a v element to this parser's ContentHandler
      ContentHandler cH=this.getContentHandler();
      cH.startElement("","v","v",atts);
  // send the entire accumulated buffer as argument to characters()
      cH.characters(verseBuff.toString().toCharArray(),0,verseBuff.length());
      cH.endElement("","v","v");
    }
    verseBuff.setLength(0); // empty the buffer
    inVerse=false; // reset boolean flag
  } // end of endElement() callback

 public InputSource resolveEntity(String publicId, String systemId)
   throws SAXException{
  // the only entity is the DTD: we want to drop the DTD, not pass it along.
return new InputSource(new StringReader(""));
}
} // end VersePicker
%>
```

What is this `resolveEntity` method? As the comment suggests, it is called upon to produce the DTD, and we don't want the source DTD because our output will not match it. In this webapp, that's the only entity resolution required, so we can simply suppress it.

Now that we have a parser/filter class defined, we can create an instance, set its content handler, and parse. But what *is* its content handler? What does it do? It receives ready-made XML as arguments to SAX callbacks, and its job is simply to echo what it receives to a character stream—in our case, to the `out` stream of the JSP page. Let's assume that we can write such a handler (call it EchoContentHandler), and look at the rest of the code.

Process Request, Create Parser, and Parse

To create a parser, we need to know the input source and, in this case, the match string. We initialize them to default values but override them with request parameters if they are submitted by the user. (See Listing 4-9.)

Listing 4-9. Initialize Parameters and Process Request.

```
<%
  // defaults for two parameters
  String xmlUri="/dat/jb/ot.xml";
  String verseMatch="create";

  // replace default values with those from request, if any
  String val=request.getParameter("xmlUri");
  if(val.length()>0) xmlUri=val;
  val=request.getParameter("verseMatch");
  if(val.length()>0) verseMatch=val;

  // if xmlUri is not a URL, construct a file:// URL
  if(0>xmlUri.indexOf(":"))
    xmlUri="file:///"+application.getRealPath(xmlUri);
```

With the two parameters in hand, we create and configure the parser and parse, as shown in Listing 4-10.

Listing 4-10. Create a Filter Object and Parse.

```
  VersePicker versePicker=new VersePicker(verseMatch);
  versePicker.setContentHandler(new EchoContentHandler(out));
 versePicker.parse(xmlUri);
%>
```

And this is the end of pickverses.jsp, except for the promised content handler.

EchoContentHandler

We are going to be sketchy in this section for three reasons. First, the code is simple. Second, echoing an XML document using a SAX content handler is a very common exercise. It is done in multiple versions in Sun's excellent Java-XML tutorial (`http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/work/Echo01.java`). Finally, writing low-level handler code is not the best way to serialize a sequence of SAX callbacks as linear-text XML. We will show a better alternative shortly.

For all of these reasons, we will only show the echo code for `startElement()`. The code is quoted from Sun's tutorial, and it assumes that there are two low-level utilities: `emit()` and `nl()`. `emit()` takes one argument, a string, and writes it to the `out` stream, and `nl()` outputs an end-of-line marker appropriate to the system on which Java is running. The only possibly tricky detail is that, to output the double-quote character, it has to be escaped as `\"`. (See Listing 4-11.)

Listing 4-11. The `startElement()` *Callback to Echo Input Document*

```
public void startElement(String namespaceURI,
                         String lName, // local name
                         String qName, // qualified name
                         Attributes attrs)
throws SAXException
{
    String eName = lName; // element name
    if ("".equals(eName)) eName = qName; // namespaceAware = false
    emit("<"+eName);
    if (attrs != null) {
        for (int i = 0; i < attrs.getLength(); i++) {
            String aName = attrs.getLocalName(i); // Attr name
            if ("".equals(aName)) aName = attrs.getQName(i);
            emit(" ");
            emit(aName+"=\""+attrs.getValue(i)+"\"");
        }
    }
    emit(">");
}
```

With `EchoContentHandler()` in place, the code of versePicker.jsp is completed. We now proceed to introduce a tool, a Transformer class, that will be with us for rest of the book.

JAXP Transformer Object

The code of Listing 4-11 does not do anything application specific; all it does is convert XML data from one representation to another (SAX callback sequence to linear text with markup). This is an operation that does not add any value, and it shouldn't require human intelligence. All such operations should be automated, and indeed they have been. The automation tool (in Java) is a Transformer class that performs XSLT transformations. It is part of the same JAXP package as SAXParserFactory and SAXParser. MSXML 3.0, which is both a parser and a transformer, provides very similar functionality.

JAXP Transformer objects perform XSLT transformations. XSLT, as we briefly explained in Chapter 1, is a programming language to transform XML data. An XSLT program (frequently called a *stylesheet*) is itself an XML document of a special kind. A general-purpose XSLT processor receives an XML data source and an XSLT program on input and produces a result as specified by the XSLT. The result is usually XML, HTML, or plain text.

A JAXP transformer that performs XSLT transformations is created with a specific XSLT program imprinted on it. It takes one input (XML data source) and produces a result as specified by the XSLT program. Because the program has been compiled into binary codes, Transformer objects are more efficient than general-purpose XSLT processors. They are also easy to integrate with other Java objects.

Transformer and TransformerFactory

You obtain a Transformer object in much the same way as you obtain a SAX parser: create a TransformerFactory instance and call its `newTransformer()` method. The method takes an argument that specifies the XSLT program to imprint on the new Transformer object.

```
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer trans = tFactory.newTransformer("myXsltProgram.xsl");
```

The main point for us now is that `newTransformer()` can also be called with no argument at all, in which case it implements the default XSLT program. The default XSLT program performs the so-called *identity transformation*: it does not change its input XML data at all. However, if the source and the result are different representations of the same XML data, then the transformer performs the useful operation of converting from one representation to another.

Sources and Results

Both XML source and XML result (if output is XML) can be any of the three standard representations: text with markup, DOM, or SAX. More precisely, the Source interface is implemented by three classes (`DOMSource`, `SAXSource`, and `StreamSource`), and similarly for the Result interface. What this means is that the identity transformer can be used for converting XML from any one of these representations to any other.

For instance, suppose that your input source is specified by a URL, `myURI`, and that you want to apply a SAX parser to it and receive output as DOM. A common reason to do this is if your input is huge and you want only a DOM subtree

of it: you use a SAX filter as your parser, filter out the subtree you want still represented as SAX callbacks, and convert that subtree to DOM. (An example is coming up soon.) The conversion can be done in a single call on the `transform()` method of the transformer, as shown in Listing 4-12.

Listing 4-12. SAX to DOM Conversion

```
 SAXSource myInput= new SAXSource(
               myXMLFilter,
               new InputSource(myURI)
      );
  DOMResult domResult = new DOMResult();
 trans.transform( myInput, domResult);
```

To create a SAXSource object, you have to give it an InputSource as an argument. You also, optionally, specify an XMLReader to use as a parser—either (as in this sample) as a constructor argument or in a separate line of code using `setXMLReader()`. If a parser is not specified, the transformer will call `XMLReaderFactory.newInstance()` to obtain a generic parser. In either case, the Transformer object will set itself as the parser's content handler and call `parser.parse(myInput)`. All the low-level code for constructing a DOM tree out of SAX callbacks is hidden inside the transformer. Similarly, if you want to convert SAX events to text with markup, you would use the same function call, but provide `new StreamResult(out)` as its second argument. All the low-level code of our EchoContentHandler class is also hidden inside the transformer.

Actually, what's hidden inside the transformer is a TransformerHandler, and it is more than just a content handler: it's an error handler, a lexical handler, and a few other things. All the information from the input is passed onwards through the "transform" call. Another way of writing Listing 4-12 is as follows:

```
SAXTransformerFactory stFactory =
        (SAXTransformerFactory)TransformerFactory.newInstance();
TransformerHandler tHandler = stFactory.newTransformerHandler();
 // Transformer trans=tHandler.getTransformer(); // if you want to work with it
myXMLFilter.setContentHandler(tHandler);
tHandler.setResult(new StreamResult(out));
myXMLFilter.parse(myURI);
```

Convert VersePicker to Text with Markup Using Transformer

We can now replace the entire EchoContentHandler class and the code that uses it (Listing 4-10) with just a few lines of code that create and use a Transformer object, either directly or as a transformer handler. After defining VersePicker, we

can either define a transformer handler as the VersePicker's content handler, or we can proceed as in Listing 4-13.

Listing 4-13. Create a Filter Object and Parse.

```
// parse, filter and output using Transformer
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer trans = tFactory.newTransformer();
trans.transform( new SAXSource(versePicker,
                 new InputSource(xmlUri)),
                 new StreamResult(out));
```

Remember that this second style is actually doing more behind the scenes; in particular, if you want to suppress DTD information from the original xmlUri, then you should use the transformer handler and call setContentHandler explicitly.

To conclude the section on SAX filters, we will show another example, illustrating SAX to DOM conversion as sketched in Listing 4-12. (We don't know yet what to do with a DOM, but that's what the last three sections of this chapter are about.) Specifically, we will write a SAX filter that will receive a string as a request parameter, and output the `book` element in ot.xml or nt.xml whose title contains the string parameter as a substring. This example addresses a fairly common situation: your XML source is too large to hold in a DOM tree in memory, but you are interested in only a subtree of your data source. Our BookPicker example is easy to generalize for many such situations.

BookPicker

Because the code of pickbook.jsp is very similar to pickverses.jsp, we will go over only the BookPicker class and the transformer code.

The BookPicker Class

The main difference from the VersePicker is that we have to maintain a more complex state. A book's title is the text content of the first child of the `book` element. We want to output both the title and the entire book if the title contains the match string. So, we need two Booleans to maintain state: `inBook` and `inBkTLong` (`bktlong` is the name of the element that contains the "long title" of the book). `inBook` is True when we know we are in a book to be copied to output; `inBkTLong` is True when we are in a `bktlong` element that may or may not match the user-submitted match string.

A further complication is that we don't know whether or not we are in the right book until we have accumulated all of its title. Fortunately, the title is the first child of the book, and there are no attributes on a book element, so we don't have to store anything from the start element of the book. When we receive the `endElement()` callback for the title, we check to see whether it's a book to be copied. If so, we output the start element of the book, the entire BkTLong element (whose text we have accumulated in a buffer), and set the `inBook` Boolean to True as a signal to copy the rest of the `book` element.

Because we now copy the entire XML content of the book including markup, we include the `ignorableWhitespace()` callback. This method has the same parameters as `characters()`, and it signals to the receiving application the beginning and the length of a stretch of whitespace characters that can be ignored. To determine whether or not a chunk of whitespace between two elements can be ignored, the parser needs to know whether the content model of the containing element is "children-only" or "mixed." Only validating parsers are required to report ignorable whitespace, but nonvalidating parsers may also use this method if they are capable of parsing and using content models.

The code for the BookPicker class is shown in Listing 4-14.

Listing 4-14. BookPicker

```
class BookPicker extends XMLFilterImpl  {
  boolean inBook=false; // true while we're in a selected book
  boolean inBkTLong=false; // true while we're in a "bktlong" element
  StringBuffer titleBuff=new StringBuffer();
  String stringMatch="";
  Attributes atts=new AttributesImpl(); // always empty in this DTD

  public BookPicker(String match) // the constructor
      throws ParserConfigurationException,SAXException {
    super(SAXParserFactory.newInstance().newSAXParser().getXMLReader());
    stringMatch=match;
    inBook=false;
    inBkTLong=false;
  }
  public void ignorableWhitespace(char[] ch,int start,int length)
      throws SAXException{
    if(!inBook) return;
    getContentHandler().ignorableWhitespace(ch,start,length);
  }
  public void characters(char[]ch,int from, int len)throws SAXException{
    if(inBook) // pass on
      getContentHandler().characters(ch,from,len);
```

```
    else if(inBkTLong) // accumulate
        titleBuff.append(ch,from,len);
  }

  public void startElement(String nsURI,String localName,String qName,
                    Attributes atts)throws SAXException{
    if(inBook) // pass on
      getContentHandler().startElement(nsURI,localName,qName,atts);
    else // set boolean value
      inBkTLong="bktlong".equals(qName);
  } // end startElement()

  public void endElement(
     String nsURI,
     String localName,
     String qName) throws SAXException{
    if(inBook){ // pass on
      getContentHandler().endElement(nsURI,localName,qName);
      if("book".equals(qName)) // end of book reached
        inBook=false;
    }
    else if(inBkTLong){ // end tag of BkTLong found; decision time
    // set inBook boolean
      inBook= (0 <= titleBuff.toString().indexOf(stringMatch));
      if(inBook){ // a match found; output start tag of book and the title
        ContentHandler cH=getContentHandler();
        cH.startElement("","book","book",atts);
        cH.startElement("","bktlong","bktlong",atts);
        cH.characters(titleBuff.toString().toCharArray(),0,titleBuff.length());
        cH.endElement("","bktlong","bktlong");
      }
      inBkTLong=false; // we are out of BkTLong
      titleBuff.setLength(0); // reset buffer
    } // end if(inBkTLong)
  }   // end endElement()
} // end BookPicker
```

Code for Output

Code for output uses the transformer. It is virtually the same as in pickverses.jsp, except one variable is renamed and we output to a DOM rather than a character stream.

```
String xmlUri="/dat/jb/ot.xml";
String bookMatch="EXODUS";
// replace default values with those from request, if any
String val=request.getParameter("xmlUri");
if(val.length()>0) xmlUri=val;
val=request.getParameter("bookMatch");
if(val.length()>0) bookMatch=val;

if(0>xmlUri.indexOf(":"))
  xmlUri="file:///"+application.getRealPath(xmlUri);

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer trans=tFactory.newTransformer();
BookPicker bookPicker = new BookPicker(bookMatch);
DOMResult domResult = new DOMResult();
SAXSource saxSource= new SAXSource(bookPicker), new InputSource(xmlUri));
trans.transform(saxSource,domResult);
Node node = domResult.getNode();
```

Once the book is in DOM, we can do much more elaborate processing because the entire tree structure is there, rather than small bits of it saved in Boolean variables. This is the subject of the DOM sections of this chapter. Before we get to them, we will have one more section on SAX to show how a SAX parser can be used to convert non-XML data to XML. This is fairly advanced stuff, so please slow down and concentrate.

SAX Parsing for Non-XML Data

We will walk through two examples in this section. For the first example, we will go back to our TagCount program from earlier in this chapter and redo its output code. In the earlier version, the output is performed by Java code that writes out tags and hashtable content to a character stream. We will replace it by a more principled piece of code that converts a hashtable to a SAX representation of XML data and uses the transformer to serialize it as text with markup. This is obviously a pedagogical example: we want to reuse familiar code to introduce a new idea. Our second example will apply the new idea to a useful task: convert fixed-width text records to XML.

Hashtable Parser for XML Output

The output code of the first version of the program is in Listing 4-7, repeated here as Listing 4-15.

Listing 4-15. Code to Display Parser Output in Text Area, Repeated from Listing 4-7

```
// output the contents of the Hashtable
%><textarea rows="30" cols="90">
<ht:Hashtable xmlns:ht="http://www.n-topus.com/namespaces/ht">
<%
 for(java.util.Enumeration e=ht.keys();e.hasMoreElements() ;){
   Object key=e.nextElement();
   Object val=ht.get(key);
%><ht:item key="<%= key %>" ><%= val %></ht:item>
<% } %>
</ht:Hashtable>
```

This code wraps the entire hashtable into a `Hashtable` element and each key-value pair into an `item` element, both within the `n-topus.com/namespaces/ht` namespace. The problem is that, once XML data is dumped to a stream like that, it is difficult to subject it to further processing. In this section, we show how to convert (language-specific) hashtable data to (language-independent) XML data, so it can be further processed or exchanged with other applications. Note that we are replacing only the code of Listing 4-15 (repeating Listing 4-7); the rest of TagCount remains unchanged.

The XML representation for our hashtable data will be a sequence of SAX procedure calls. Remember that the calls going from the parser to the content handler carry all the data from the parsed document. Moreover, they show its hierarchical structure in the temporal sequence of `startElement()` and `endElement()` calls. So, we are going to use a SAX "parser" to make a hashtable look like XML.

The process of SAX parsing, as we said from the beginning, is a conversation between a parser and a content handler. In the examples so far, the parser is something that somebody else wrote, at Apache or Microsoft or Sun. It is a program that knows, for instance, how to collect all material between a left bracket and a right bracket, break it into the tag name and attributes, recognize namespaces, package all that information into three strings and an Attributes object, and call `handler.startElement()`. Now we are going to write a parser ourselves. It will be a very simple parser; it will "parse" a hashtable. Specifically, for each key-value pair in the hashtable, it will perform the tasks as described in the following paragraph.

It will start by saying: “Hey, I have found the start of a new element, and so I am going to call `handler.startElement()`. The element’s prefix is `ht` mapped to `n-topus.com/namespaces/ht`. Its local name is `item`. Its qName is `ht:item`. It has one attribute that is such and so; its value is the key of the key-value pair. It has text content that is the value corresponding to that key. And now I have found the end of that element, and so I’m going to call `handler.endElement()`.” Together, all these calls will translate the contents of the hashtable into arguments to SAX procedure calls. The handler that responds to those calls can do any processing whatsoever. In our case here, the handler is the default transformer that will convert its SAX source to text with markup, echoed to a character stream.

Schematically, the main components of TagCount2.jsp are shown in Figure 4-5.

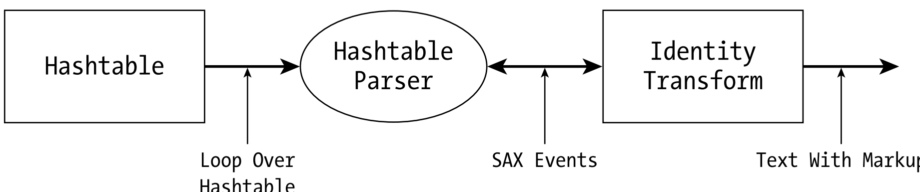

Figure 4-5. Hashtable parser and transformer

Hashtable Parser Code

We are going to define a class that implements the XMLReader interface. The way it works in Java is that, if a class implements an interface, it has to implement all its methods, even if the application needs only one or two of them. So, a big part of our parser consists of do-nothing implementations of required methods. The only method we really have to work on is `parse()`.

XMLReader Methods Other Than `parse()`

Before we get to implementing methods, we declare the variables and define a constructor, as shown in Listing 4-16.

Listing 4-16. Variables and a Constructor for HashtableReader

```
class HashtableReader implements XMLReader{
 // read from the hashtable
    ContentHandler handler;
    ErrorHandler errHandler;
    Hashtable ht;
    public HashtableReader(Hashtable ht){this.ht=ht;}
```

Now, the required methods. We will use this occasion to review the XMLReader interface. Because we don't really need these methods, we provide only minimal implementations for them, as shown in Listing 4-17.

Listing 4-17. Minimal Implementation of the XMLReader Interface

```
// set and get the handlers, including EntityResolver
public void setContentHandler(ContentHandler h)  {handler=h;}
public ContentHandler getContentHandler() {return handler;}
public void setErrorHandler(ErrorHandler h)  {errHandler=h;}
public ErrorHandler getErrorHandler()  {return errHandler;}
public void setDTDHandler(DTDHandler handler)  {}
public DTDHandler getDTDHandler()  {return null;}
public void setEntityResolver(EntityResolver resolver)  {}
public EntityResolver getEntityResolver()  {return null;}

// set and get Features and Properties
public void setFeature(String name, boolean value)  {}
public boolean getFeature(String name)  {return false;}
public void setProperty(String name, Object value)  {}
public Object getProperty(String name)  {return null;}

// two required parse() methods: from InputSource or systemID
public void parse(InputSource input) throws SAXException {parse();}
public void parse(String systemId) throws SAXException {parse();}
```

The XMLReader interface specifies that an implementing class must have two versions of the `parse()` method, one taking an InputSource argument, the other a string that is a URL (`systemID`). Our little parser doesn't really need either of them because its input comes from the hashtable via an enumeration loop. So the method that is going to do the real work is `parse()` that takes no arguments. (We could, in fact, call it anything, but `parse()` is a good name.) The required dummy versions simply call the real `parse()`.

The Real Parse

Strictly speaking, we should configure our parser's properties and have it behave accordingly, but, because we're talking to ourselves, we can skip the formalities and simply assume that our parser is namespace aware but reports xmlns declarations as attributes. Being namespace aware, it must mark the boundaries of scope of each namespace declaration by calling two methods: `startPrefixMapping()` and `endPrefixMapping()`. It also reports all three strings to the handler: the local name, the namespace URI, and the qName. In creating an

attribute, it lists the namespace URI (which is empty), the local name, the qName, the data type (CDATA), and the value.

Listing 4-18 shows the `parse()` method that actually gets called by HashtableParser:

Listing 4-18. The Real `Parse()` *Method of HashtableParser*

```
    public void parse()throws SAXException{
      String htURI="n-topus.com/namespaces/ht";
      handler.startDocument();
  // start prefix mapping immediately
  // because namespace is declared on the root element
      handler.startPrefixMapping("ht",htURI);
      AttributesImpl attr=new AttributesImpl();
      attr.addAttribute("xmlns","ht","xmlns:ht","CDATA",htURI);
      handler.startElement(htURI,"Hashtable","ht:Hashtable",attr);
      attr.clear();
      attr.addAttribute("","key","key","CDATA","dummy");
      for(java.util.Enumeration e=ht.keys();e.hasMoreElements() ;){
        Object key=e.nextElement();
        Object val=ht.get(key);
        attr.setAttribute(0,"","key","key","CDATA",key.toString());
        handler.startElement("ht","item","ht:item",attr);
        String str=val.toString();
        handler.characters(str.toCharArray(),0,str.length());
        handler.endElement("ht","item","ht:item");
      }
      handler.endElement("ht","Hashtable","ht:Hashtable");
      handler.endPrefixMapping("ht");
      handler.endDocument();
    }
 } // end of HashtableReader
```

Converting HashtableReader to Text with Markup

To serialize HashtableReader, we use a transformer. Because we want to convert from SAX to text with markup, we set up the source and the result accordingly. In particular, the source is a SAXSource that is created with two arguments: the parser, which is a hashtable reader, and an input source. The transformer sets itself as a handler and calls the parser's `parse()` method, giving it the input source as an argument. The tricky part is that our XML input data is not really XML, and the `parse()` method that we want to be called takes no arguments.

However, recall that we have implemented the standard `parse()` method with an InputSource argument so that it ignores its argument and calls our "real parse":

```
public void parse(InputSource input) throws SAXException { parse(); }
```

So, in our transformer code, we create the required dummy InputSource, call the standard `parse()` method, and our real parse does get called, as shown in Listing 4-19.

Listing 4-19. Parser, Transformer, and InputSource

```
// create parser, parse XML data input into a Hashtable that counts keys
SAXParserFactory factory=SAXParserFactory.newInstance();
 if("yes".equals(request.getParameter("namespaces")))
   factory.setNamespaceAware(true);
 factory.newSAXParser().parse(iS,keyCounter);
 // info about the document is now stored in the hashtable;
 // create HashtableReader
   HashtableReader hashtableReader=new HashtableReader(ht);
%><textarea rows="30" cols="90">
<%
// new code begins
   TransformerFactory tFactory = TransformerFactory.newInstance();
// create a Transformer for identity transformations
   Transformer trans = tFactory.newTransformer();
// configure the Transformer to skip XML declaration
// (useful if want to insert output in other XML data)
// and indent output to reflect tree structure
   trans.setOutputProperty(
     OutputKeys.OMIT_XML_DECLARATION,"yes");
   trans.setOutputProperty(
     OutputKeys.INDENT,"yes"
   );
// create a dummy InputSource
   InputSource dummy= new InputSource();
   trans.transform( // takes two arguments, Source and Result
      new SAXSource(
               hashtableReader,   // our intended parser
              dummy
      ),
      new StreamResult(out)
   );
 %>
</textarea>
```

Behind the scenes, `hashtableReader.parse(dummy)` is called, which in turn calls the real parser to convert the internal hashtable into a sequence of SAX callbacks. The transformer converts that sequence into a marked-up text sent down the `out` stream, so the text ends up in the text area.

Fixed-Width Records

HashtableParser is something of a pedagogical example that reused an earlier example to introduce a new idea. We can now apply that new idea to a useful task: converting fixed-width text records to XML. We will again use an XMLReader to "parse" each individual record sending out SAX callbacks, and we will use a Transformer object to serialize the callbacks as marked-up text. Figure 4-6 shows FixedWidth.jsp in action.

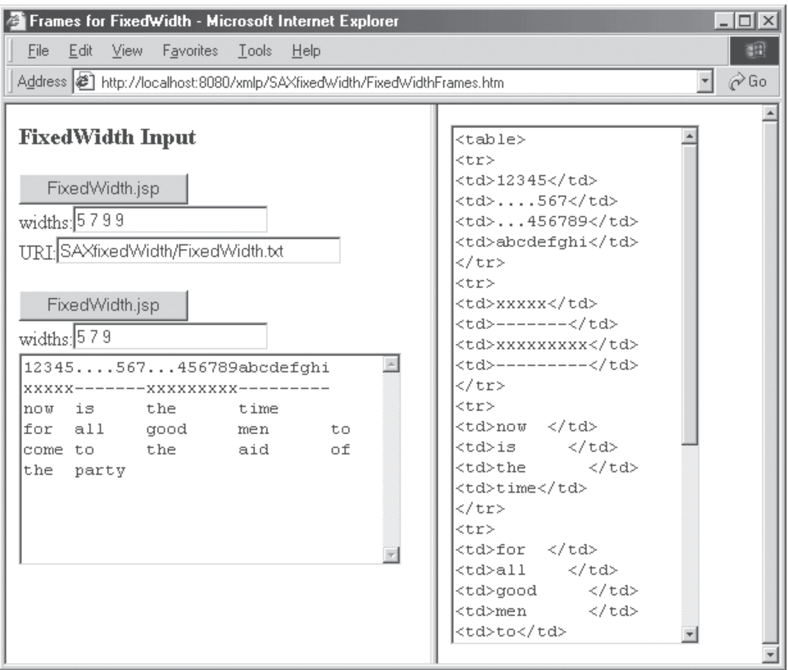

Figure 4-6. Converting fixed-width records to XML

To perform the conversion, we need to know the widths, and these are provided as space-separated tokens. One of the components of the program converts them to integers.

The program outputs an XML document conforming to the following DTD:

```
<!DOCTYPE table [
<!ELEMENT table (row*)>
<!ELEMENT row (td*)>
<!ELEMENT td (#PCDATA)>
]>
```

The program is actually more flexible than Figure 4-6 suggests. First, the number of widths does not have to equal the number of fields. If there are fewer widths than fields, all the extra fields are assumed to have the last width. (So, the input box for widths could have just three tokens, “5 7 9”, with no change in functionality.) Second, the number of fields does not have to be the same in all records. The program can output rows containing different numbers of data elements.

Program Outline

In outline, the program consists of five steps:

1. Import libraries.
2. Define a function to convert widths to integers.
3. Define the parser class, FixedWidthReader (similar to HashtableReader).
4. Process request parameters, and initialize input source, the integer array of widths, and a FixedWidthReader.
5. Create a transformer that takes a FixedWidthReader as its source and outputs marked-up text.

The biggest difference between this program and the HashtableReader is that, in this case, our parser has to process an actual input source, a character stream that may come from a text file or a text area in an HTML page. Because the programs are otherwise very similar, we will present the entire FixedWidth.jsp in one go in Listing 4-20, but leaving stubs for the text-to-integers conversion function and the FixedWidthReader class.

Listing 4-20. FixedWidth.jsp, with Gaps

```
<%@ page errorPage="error.jsp"
    import=" . . . "
   // import from javax.xml, org.xml.sax, java.io and java.util packages
%><%!
// Convert string of tokens to integer array
  public static int[] intArray(String str){
    // see separate section
  }
%><%
 class FixedWidthReader implements XMLReader{
 // Read from lines of fixed-width data as if from html table.
 // See separate section
}
// Process request,
 // initialize InputSource, array of widths and FixedWidthReader
 String docStr=request.getParameter("doc");
 String uriStr=request.getParameter("uri");
 InputSource iS;
 if(docStr!=null) // input from textarea
    iS=new InputSource(new StringReader(docStr));
 else
    iS=new InputSource(application.getRealPath(uriStr));
 int[]widths=intArray(request.getParameter("widths"));
 FixedWidthReader fixedWidthReader=new FixedWidthReader(widths);
%><textarea rows="30" cols="60">
<%
// Parse, transform and dump output into textarea
   TransformerFactory tFactory = TransformerFactory.newInstance();
   Transformer trans = tFactory.newTransformer();
   trans.setOutputProperty(
     OutputKeys.OMIT_XML_DECLARATION,"yes");
   trans.setOutputProperty(
     OutputKeys.INDENT,"yes"
   );
     trans.transform(new SAXSource(
        fixedWidthReader, iS),      // parse InputSource using FixedWidthReader
        new StreamResult(out));  // convert to marked-up text, send to out stream
%>
</textarea>
```

String of Tokens to Integer Array

We want to convert a string like "9 7 23" to an integer array. Our problem is that Java arrays are fixed length, and we don't know in advance how many tokens to expect. We have to use a dynamic array (vector) and, when finished, check its length and create an integer array of that length. However, Java vectors can contain only objects and not primitive data types such as integer or Boolean. So, we will be converting string tokens to Integer objects, to put them in the vector, and convert from Integer objects to plain integers when creating an array. (See Listing 4-21.)

Listing 4-21. String to Integer Array

```
public static int[] intArray(String str){
 Vector intVec=new Vector();
 StringTokenizer st=new StringTokenizer(str);
 while (st.hasMoreTokens())
// get next token, create an Integer object, add to Vector
    intVec.add(new Integer(st.nextToken()));
 int[] res=new int[intVec.size()];
 for(int i=0;i<res.length;i++)
// extract plain integer from an Integer object
   res[i]=((Integer) intVec.get(i)).intValue();
 return res;
}
```

FixedWidthReader

The FixedWidthReader class is quite similar to HashtableReader and has the same basic outline.

1. Declare variables and define constructor.
2. Define required "do-nothing" methods.
3. Define parse methods.

We will show the first and the last step only, because the middle step is identical in both classes.

Variables, Constructor, Methods Other than `parse()`

Variables, as before, include the required handlers and the program-specific variable to hold the array of widths. The constructor initializes that variable, as shown in Listing 4-22.

Listing 4-22. Variables, the Constructor

```
  class FixedWidthReader implements XMLReader{
    ContentHandler handler;
    ErrorHandler errHandler;
    AttributesImpl attr=new AttributesImpl();
    protected int[]widths=null; // class-specific array of widths
    public FixedWidthReader(int[]w){widths=w;}  // constructor

// required methods declared in XMLReader (see HashtableReader):
// set and get methods for handlers, features and properties
```

Parse Methods

As we said, the `parse()` method has two versions: one takes an InputSource argument, and the other a string that is a URL (`systemID`). HashtableReader has both of them call its own `parse()` of no arguments. In this program, we actually implement the InputSource version; the URL version calls it with an InputSource created from system ID. (See Listing 4-23.)

Listing 4-23. Parse Method with a System ID Argument

```
    public void parse(String systemId) throws SAXException {
      parse(new InputSource(systemId));
    }
```

The InputSource method, in conformance with the SAX specification, checks to see whether its input comes from a character stream, a byte stream, or a URL (in that order). (Because we are parsing our own input, we know that it comes either from a character stream or from a text file, but the specification mandates the test.) If the input is a byte stream, it is converted to a character stream using a given encoding. Whatever the input source, we end up with a character stream that delivers lines of text with fixed-width data. The stream must be capable of counting lines, in case the user requires information on the specific location of events; in Java, such a stream class is called LineNumberReader.

With a character stream in hand, we can start calling handler methods. Our `parse()` method actually makes only four calls: a `start/endDocument()` pair and one `start/endElement()` pair to output the `root` table element. The hard work of

processing each line is parceled out into a separate method, `parseRow()`. (See Listing 4-24.)

Listing 4-24. Parse Method with an InputSource Argument

```
  public void parse(InputSource iS) throws SAXException {
    LineNumberReader reader;
    try{ // a InputSource parser must try these options in this order
     if(null!=iS.getCharacterStream())
       reader=new LineNumberReader(iS.getCharacterStream());
     else if(null!=iS.getByteStream()){
       String encoding=iS.getEncoding();
       if(encoding==null)encoding="utf-8"; // the default
       reader=new LineNumberReader(
                   new InputStreamReader(iS.getByteStream(),encoding));
     }
     else reader= new LineNumberReader(new FileReader(iS.getSystemId()));
    }catch(IOException ioe){ // catch IO exception, output message
      throw new SAXException(ioe.getMessage()+" on "+iS.getSystemId());
    }
  // start callbacks
    handler.startDocument();
    handler.startElement("","table","table",attr);
    try{
    for(String line=reader.readLine(); line!=null; line=reader.readLine())
      parseRow(line.toCharArray());
  }catch(IOException ex){
   throw new SAXException(ex.getMessage()+"; line "+reader.getLineNumber());
  }
  handler.endElement("","table","table");
  handler.endDocument();
}
```

Finally, `parseRow()` receives each line as a character array and works through the line outputting each field as a `<td>` element. It uses two integer variables, `lo` and `w`, to hold the starting position and the width of each field. Together with a character array, these are precisely the arguments to the `characters()` callback: a character array, the start point, and the number of characters to send. (See Listing 4-25.)

Listing 4-25. Parse Each Line

```
protected void parseRow(char[]ch) throws SAXException{
  if(0==ch.length) return;//empty row
  int i=0,   // index into the widths array
  lo=0, // beginning of next field
  w=widths[0]; // width of next field
  handler.startElement("","tr","tr",attr);
  while(lo<ch.length){
    int nextLo=lo+w;
    int excess = nextLo - ch.length; // for processing the last column
    if(excess > 0){ // the last column is not its complete width
      nextLo=ch.length; w=w-excess;
    }
  // output the next field as a <td> element
    handler.startElement("","td","td",attr);
    handler.characters(ch,lo,w);
  // includes fixed-width extra blanks; can be trimmed here
    handler.endElement("","td","td");
  // increment the index into width array unless it's at the last element
    if(i!=widths.length-1) i++;
    lo=nextLo;
    w=widths[i];
  }
  handler.endElement("","tr","tr");
}
```

This concludes the fixed-width Java code, and the entire SAX section. As an exercise, we suggest that you write a similar program for tab-separated fields and for CSV (comma-separated values) data. In the remainder of the chapter, we introduce DOM and show how DOM and SAX can usefully work together.

DOM Programming

DOM provides standard interfaces for working with data structures that represent XML data. The DOM recommendation says nothing about how the data structures are implemented, or even in what language: they provide only APIs. These APIs strongly imply that the data structures form a tree of nodes, similar but not identical to the XPath tree. (We will list the differences shortly.) Beginning with *DOM 3* and *XPath 2.0*, the tree structures of DOM and XPath will be fully compatible, both conformant with the *Infoset* recommendation. This will make it easier to standardize DOM interfaces that use XPath expressions.

To give you a brief preview, the main DOM interfaces are Node and Document. The Document interface is derived from Node; that is, a document is a kind of node and has all its properties and methods. Node objects have a type property that takes the familiar values of `element`, `attribute`, `comment`, and so forth. A common pattern of DOM programming is to traverse the entire tree visiting each node, and do to each node according to its type. This is frequently done in a recursive fashion:

1. Initially set the current node to the document object.

2. If the current node is Null, return.

3. Otherwise, visit and process the current node.

4. For each child of the current node, set the current node to that child and make a recursive call.

A DOM application typically begins by obtaining an instance of a DOM parser and creating a DOM document object. You have seen examples of how this is done in JSP and ASP in earlier chapters.

A Brief History

The notion of the Document Object Model (or *DOM*) first became widely known with the release of fourth-generation browsers, as part of dynamic HTML. Within that context, DOM meant a set of naming conventions and APIs for working with objects in an HTML page. Its area of application was the Web browser.

Although DOM was supposed to be language independent and standard across browsers, in practice the two major browsers simply implemented their DOMs as they wanted them to be, in JavaScript and, in the case of Microsoft, also in VBScript. A more narrowly defined HTML DOM, without an event model, was codified by W3C as *DOM Level 0* in late 1997. The IE4 DOM was very close to that specification, whereas the NC4 DOM, released a few months earlier, was substantially different.

DOM Level 1 was released in October 1998. Its main innovation was to add a DOM specification for XML. *DOM Level 1* consists of three parts:

- Fundamental interfaces that are common to XML and HTML and must be implemented by all DOM-compliant processors, including XML parsers and HTML browsers. Fundamental interfaces specify the structure and behavior of Document, Node, and other fundamental structural elements of any HTML or XML document.

- Extended interfaces that apply only to XML. They specify those items that are never found in an HTML document: DTD, processing instructions, entities and entity references, and so on.

- HTML-specific interfaces that have to do with specific HTML elements and the event model.

Level 1 groups fundamental and extended interfaces together as Core interfaces. They remain so grouped in the current version, *DOM Level 2*, released in November 2000. As before, a compliant XML processor (DOM parser) must implement Core interfaces in their entirety. In addition, *DOM Level 2* specifies several optional groups of interfaces: Views, Styles, Events, Traversal, and Range. To promote modularity, each group is defined in a separate specification, except that the Traversal and Range groups are covered in a single specification, probably for bureaucratic reasons. (The same group of people wrote both.) Otherwise, Traversal and Range have little in common: Traversal interfaces are for visiting all nodes of the tree in a uniform way, and Range interfaces have to do with ranges of text between two endpoints and are mostly useful in the GUI situation of a browser or an XML editor. The same is true of the Views, Styles, and Events specifications: like Range, they apply, at least for now, to GUI situations only, and we are not going to discuss them in any detail. In the remainder of the book, we will be using Core and Traversal interfaces.

Testing for Optional Interfaces

Because Traversal interfaces are optional, you need a way to test whether they are implemented by your processor. The `DOMImplementation` interface of the Core declares the `hasFeature(feature, version)` method precisely for this purpose. Run it with parameter values "Traversal" and "2.0" to determine whether or not (respectively) this module is supported. It is, indeed, supported both by Apache Xerces and MSXML.

DOM Interfaces Overview

DOM specifies interfaces that are to be implemented in some programming language. They are specified in a language-independent way using a formal notation called, reasonably enough, *Interface Definition Language* (*IDL*). Several such languages exist, including one that Microsoft uses to specify COM interfaces, and another one from Object Management Group (OMG) that is used to specify CORBA interfaces. W3C uses the OMG language but makes it clear that the choice does not imply any kind of taking sides in the COM-CORBA contest.

DOM interfaces written in IDL are twice removed from the actual implementation. To get to an implementation, you first have to choose a programming language and translate DOM interfaces into the appropriate constructs of that language: abstract classes in C++, interfaces in Java, objects and properties in JavaScript, and so on. This process is called *language binding:* the language-independent interfaces of DOM are bound to constructs in a specific language. No actual code is written in the process, only declarations. In the second stage of implementation, the language-specific constructs, such as Java interfaces, are implemented in actual working code. That code is used in application programming.

To ensure that different DOM implementations in at least some languages are compatible with each other, W3C itself provides the first stage of implementation—language binding—for two languages, Java and JavaScript (ECMAScript). Each DOM specification contains three appendices (among others) that contain IDL definitions, Java language binding, and ECMAScript language binding. We will look at an example of the same interface in IDL, Java, and ECMAScript after we review the Node interface and node types. We are not presenting a complete listing of the entire DOM, but the interfaces we do present and the code that uses those interfaces provide enough background to learn the remaining ones as needed by simply perusing the APIs.

Node and Node Types

The central DOM interface is Node, from which a number of other interfaces are derived. The central design feature of DOM is that every type of XML data—element, attribute, comment, PI, CDATA section, and so on—is represented by a Node object of a specific type. Specifically, this means that DOM Core defines Element, Attribute, Comment (and so on) interfaces that are all derived from Node and that inherit its attributes and methods. In addition, the DOM Core defines an integer constant for each node type, as shown in Table 4-2.

Table 4-2. DOM Node Types

CONSTANT NAME	CONSTANT VALUE	INTERFACE NAME
ELEMENT_NODE	1	Element
ATTRIBUTE_NODE	2	Attr
TEXT_NODE	3	Text
CDATA_SECTION_NODE	4	CDATASection
ENTITY_REFERENCE_NODE	5	EntityReference
ENTITY_NODE	6	Entity
PROCESSING_INSTRUCTION_NODE	7	ProcessingInstruction
COMMENT_NODE	8	Comment
DOCUMENT_NODE	9	Document
DOCUMENT_TYPE_NODE	10	DocumentType
DOCUMENT_FRAGMENT_NODE	11	DocumentFragment
NOTATION_NODE	12	Notation

Whereas types 1 through 9 represent the familiar information items, types 10 through 12 require a brief comment, for the sake of completeness.

DocumentType (not to be confused with a node type) is an interface for working with DTDs. It provides very limited functionality, representing the DTD, its internal subset, and the SYSTEM and PUBLIC identifiers simply as strings. There is a working draft of a *DOM Level 3* module (`http://www.w3.org/TR/DOM-Level-3-ASLS/abstract-schemas.html`) that will provide elaborate APIs for working with "abstract schemas" such as a DTD or an XML schema.

DocumentFragment is an interface for "lightweight" tree objects designed for such operations as cutting and pasting a tree of nodes, or splicing a sequence of nodes. There is a candidate recommendation (`http://www.w3.org/TR/xml-fragment`) that defines what can constitute a fragment, what context information is needed to move a fragment into a different document, and how to define such context information. It is unlikely, however, that you will see Fragment nodes until this specification is completed and integrated with DOM.

Notation is an interface for very rarely used XML notations. You are very unlikely to come across Notation nodes.

Overview of Node Methods

The methods of the Node interface fall into several groups:

- *Get information about the current node*: `getNodeName()`, `getNodeType()`, `getNodeValue()`, `getOwnerDocument()`, `hasChildNodes()`, `hasAttributes()`
- *Namespace support*: `namespaceURI()`, `prefix()`, `localName()`
- *Clone and modify the current node*: `cloneNode()`, `setNodeValue()`, `normalize()`
- *Modify the node's children*: `appendChild()`, `removeChild()`, `replaceChild()`, `insertBefore()`
- *Move around in the tree*: `getAttributes()`, `getChildNodes()`, `getFirstChild()`, `getLastChild()`, `getNextSibling()`, `getPreviousSibling()`, `getParentNode()`

In addition, there is an `isSupported(feature,version)` method that performs the same function as the `hasFeature()` method of the DOMImplementation interface.

All these methods are inherited by all the "node type" interfaces. One of these derived interfaces is Document. We will use it to illustrate IDL and language bindings after we briefly go over the interfaces that are not derived from Node.

Other Interfaces and Classes

The Node interface and interfaces derived from it repackage the basic XML specifications (*XML 1.0* and *XML Namespaces*) in terms of objects, properties, and methods. Because DOM is a step closer to processing than the XML specification, it includes several additional features that meet computational needs. These include

- *DOMString*: A sequence of integers that becomes a String object in language bindings.
- *DOMException*: An exception class that contains numeric codes for frequently occurring exceptions.

- *NodeList and NamedNodeMap*: Two interfaces for dealing with collections of nodes. NodeList is a dynamically resizable array of nodes that is used to represent the children of a node and other ordered collections of nodes. NamedNodeMap is an associative array of name-value pairs; it is used to represent a node's attributes.

Example: Document Interface in IDL, Java, and ECMAScript

We start with the IDL code (Listing 4-26), which is similar to C++ and largely self-explanatory. The term *attribute* is used to describe what would be called an instance variable in Java. Additional explanations are provided in comments. Several times, the code mentions the NodeList interface.

Listing 4-26. The Document Interface in IDL

```
interface Document : Node { // interface Document is derived from Node
  readonly attribute DocumentType doctype;
  readonly attribute DOMImplementation  implementation;
  readonly attribute Element documentElement; // the root of the element tree
  Element  createElement(in DOMString tagName) raises(DOMException);

  // Omitted: similar createXX() methods where XX stands for:
  //  DocumentFragment, Text,  Comment, CDATASection,
  // ProcessingInstruction, Attribute, and EntityReference

  NodeList getElementsByTagName(in DOMString tagname);
  // Introduced in DOM Level 2:
  Node importNode(in Node importedNode, in boolean deep) raises(DOMException);
  // Introduced in DOM Level 2: support for Namespaces
  Element createElementNS(in DOMString namespaceURI, in DOMString qualifiedName)
                                      raises(DOMException);
  Attr createAttributeNS(in DOMString namespaceURI, in DOMString qualifiedName)
                                      raises(DOMException);
  NodeList getElementsByTagNameNS(in DOMString namespaceURI,
                                       in DOMString localName);
  // Introduced in DOM Level 2:
  Element getElementById(in DOMString elementId);
};
```

This is how this comes out in Java (Listing 4-27). Java interfaces cannot have instance variables, only Get/Set access methods. The `readonly` qualifier of IDL is expressed by the absence of a `setXX()` method for the corresponding variable.

Listing 4-27. The Document Interface in Java

```
public interface Document extends Node {
    public DocumentType getDoctype();
    public DOMImplementation getImplementation();
    public Element getDocumentElement();
    public Element createElement(String tagName) throws DOMException;
    // Omitted similar createXX() methods, where XX stands for:
    // DocumentFragment, Text,  Comment, CDATASection,
    // ProcessingInstruction, Attribute, and EntityReference
    public NodeList getElementsByTagName(String tagname);
    public Node importNode(Node importedNode, boolean deep) throws DOMException;
    public Element createElementNS(String namespaceURI, String qualifiedName)
                                  throws DOMException;
    public Attr createAttributeNS(String namespaceURI, String qualifiedName)
                                 throws DOMException;
    public NodeList getElementsByTagNameNS(String namespaceURI, String localName);
    public Element getElementById(String elementId);
}
```

Finally, in ECMAScript we have pseudo-code with English-language explanations because the language does not have data types. For instance,

```
getElementsByTagName(tagname)
    This method returns a NodeList object.
    The tagname parameter is of type String.
```

This completes our overview of the Core DOM APIs. It is definitely time for an example. After the example, we will introduce the Traversal APIs and redo the example using those.

Simple DOM Example: Tree Address

Our application will associate a "tree address" with each element node. We define a tree address as a sequence of dot-separated integers such that each integer

gives the 0-based child number in the next level of the tree, starting from the root element. For instance, 0.2.1 is the address of (reading from the right):

the second child of

the third child of

the first child of

the root element

In terms of DOM methods, the node with this tree address can be retrieved from a Document object named "doc" by the following call. (Note that Java syntax allows whitespace before the "dot" operator, which makes it possible to line up method calls and insert comments.)

```
doc.getDocumentElement()  // returns the root of the element tree
    .getChildNodes().item(0)     // getChildNodes() returns a NodeList
    .getChildNodes().item(2)
    .getChildNodes().item(1)
```

Concatenating the document's URL with the tree address of a node yields a globally unique identifier for that node.

Application Outline and the Main Page

Our application is "DOMtreeAddr/treeAddr.jsp" (see Listing 4-28 within the xmlp webapp), and it receives a document to process from the text area of an HTML form. You can enter and edit it manually, you can copy and paste it, or you can load it into the text area from a URL.

The application parses the document, stores the Document object in cache, and adds tree addresses to element nodes as values of the `treeAddr` attribute. We show three ways of doing this:

- using Core interfaces and a stack to traverse the document
- using Core interfaces and traversing the document recursively
- using Traversal interfaces

The user can also remove the `treeAddr` attributes from the cached Document object or remove the Document object from cache to process another one. Figure 4-7 shows all the buttons of `http://localhost:8080/xmlp/DOMtreeAddr/treeAddr.jsp`.

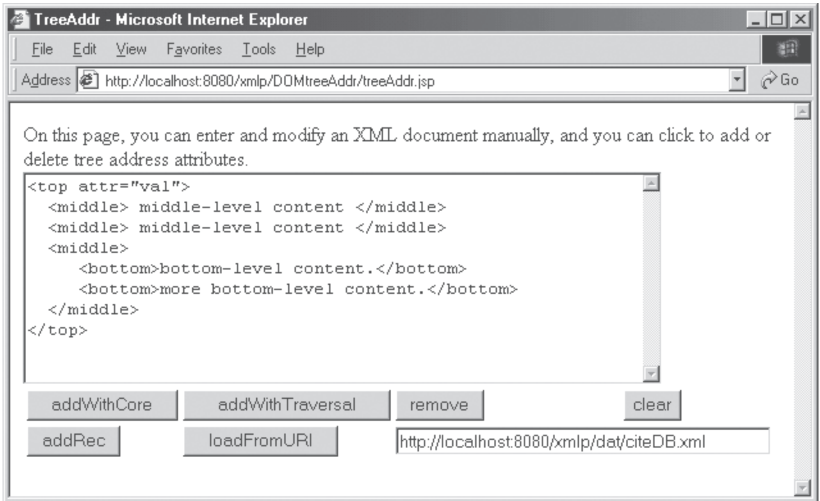

Figure 4-7. The Tree Address application

The Code of the Main Page

The code of treeAddr.jsp follows the familiar pattern:

1. Import libraries and instantiate session cache.
2. Initialize session variables and store them in cache (once per session).
3. Output the HTML form.

Let's look at the first two steps first (Listing 4-28). As in many other examples, the session variables are a DocumentBuilder (that is, a DOM parser) and a transformer that implements the identity transformation. The parser is used every time during a session when a new XML document needs to be parsed. The transformer is used to serialize a Document object to an output stream that, in turn, dumps the serialized document into the text area.

Listing 4-28. Page Directive and Session Variables

```
<%@ page errorPage="error.jsp"
    import="javax.xml.transform.*,
            javax.xml.transform.stream.*,
            javax.xml.transform.dom.*,
            org.w3c.dom.*,
            javax.xml.parsers.*"
%><jsp:useBean id="sessCache" class="java.util.Hashtable" scope="session"
/><% // initialize session variables if they're not already set up
 DocumentBuilder db=(DocumentBuilder) sessCache.get("db");
 if(db==null){
   DocumentBuilderFactory dbf=DocumentBuilderFactory.newInstance();
   dbf.setNamespaceAware(true);
   db=dbf.newDocumentBuilder();
   sessCache.put("db",db);
}
 Transformer trans=(Transformer) sessCache.get("trans");
 if(trans==null){
   TransformerFactory tFactory = TransformerFactory.newInstance();
   trans = tFactory.newTransformer();
   trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,"yes");
   trans.setOutputProperty(OutputKeys.INDENT,"yes");
   sessCache.put("trans",trans);
}
```

The HTML form has only one small piece of code in it: we check to see whether the current document has already been parsed and the resulting Document stored in the cache; if so, we dump it into the text area for display. Otherwise, we output a default document to work with:

```
<textarea name="doc" rows="30" cols="80">
<%
  Document doc=(Document)sessCache.get("doc"); // do we have a document?
  if(doc!=null) // yes: dump it into the text area
    trans.transform(new DOMSource(doc),new StreamResult(out));
  else {
%><!-- output an XML document here -->
<%  }  // close the else branch
%></textarea>
```

The rest of the form outputs six buttons, each of which calls a JSP page. We will concentrate on the three addXXX.jsp pages because the rest of them do not add any new material.

Add Tree Addresses Using Core Interfaces

Our task is to traverse the DOM tree computing tree addresses for element nodes as we go along. Ordinarily, you would perform this task using a Traversal interface. However, it is useful to see how traversal is done using only the low-level Core interfaces, especially because they predate Traversal interfaces by a couple of years and a lot of installed code relies on them for traversal.

As we mentioned, we show two versions: one that uses a stack to keep track of nodes to visit, and another that uses recursion. Listing 4-29 shows the stack version.

Listing 4-29. Stack-Based `addTreeAddr()` *from addTreeAddrCore.jsp*

```
  public void addTreeAddr(Document doc) {
    java.util.Stack stack = new java.util.Stack();
// push the root of the element tree on stack
    stack.push(doc.getDocumentElement());
    do{  // repeat until stack is empty
       Element elt=(Element)stack.pop();
     // Java Stacks hold objects of type Object
     // that need to be cast to a specific type.
     // We put only Element objects on our stack
       String treeAddr=elt.getAttribute("treeAddr");
    // if the attribute does not have a value
    // getAttribute() returns the empty string
       if(treeAddr.length()==0) // the root element
         elt.setAttribute("treeAddr","");
       else treeAddr+=".";
       NodeList nodeList=elt.getChildNodes();
       for(int i=0;i<nodeList.getLength();i++){
         Node child=nodeList.item(i);
         if(child.getNodeType()!=org.w3c.dom.Node.ELEMENT_NODE)
            continue; // skip non-element nodes
         ( (Element)child).setAttribute("treeAddr",treeAddr+i);
          // for 1-based addresses, we'd say:
          // setAttribute("treeAddr",treeAddr+(i+1))
         stack.push(child);
       }
    } while(!stack.empty());
  }
```

The recursive version of this procedure, shown in Listing 4-30, takes two arguments: the current node and its tree address (a string).

Listing 4-30. Recursive `addTreeAddr()`*, from addTreeAddrRec.jsp*

```
public void addTreeAddrRec(Node node, String treeAddr){
    if(node.getNodeType()!=org.w3c.dom.Node.ELEMENT_NODE)
      return; // If it's not an Element, do nothing.
    Element elt=(Element) node;
    elt.setAttribute("treeAddr",treeAddr);
    if(treeAddr.length()>0) treeAddr+=".";
    NodeList nodeList=node.getChildNodes();
    for(int i=0;i<nodeList.getLength();i++)
      addTreeAddrRec(nodeList.item(i),treeAddr+i);
  }
```

We start the recursive process with this line of code:

```
addTreeAddrRec(doc.getDocumentElement(),"");
```

Apart from this line, the rest of the page is identical for the stack and the recursive versions, as shown in Listing 4-31.

Listing 4-31. Add-Address Page Using Core Interfaces (Stack or Recursive)

```
<%@ page errorPage="error.jsp"
    import="org.w3c.dom.*,  javax.xml.parsers.*"
%><jsp:useBean id="sessCache" class="java.util.Hashtable" scope="session"
/>

<%! /* the definition of the appropriate addAddr procedure goes here */ %>

<%
 String docStr=request.getParameter("doc");
 DocumentBuilder db=(DocumentBuilder) sessCache.get("db");
 if(docStr==null || db==null){  // user bypassed the entry page
%>  <jsp:forward page="treeAddr.jsp"/>
<% }
 Document doc=db.parse(
                 new org.xml.sax.InputSource(
                    new java.io.StringReader(docStr)));
 addTreeAddr(doc);  // this is the stack version;
 // addTreeAddrRec(doc.getDocumentElement(),""); // the recursive version

 sessCache.put("doc", doc);

%><jsp:forward page="treeAddr.jsp"/>
```

Traversal Interfaces

DOM Level 2 defines two kinds of objects that can be attached to a tree of nodes in order to traverse it: NodeIterator and TreeWalker. NodeIterators are better suited for working with text content of each visited node, whereas TreeWalkers are better suited for working with document structure. We will sometimes say *traverser* to refer to either a NodeIterator or a TreeWalker.

Both kinds of traversers present a "logical view" of the XML data they traverse that may include only a subset of all the nodes in the data. We will say that nodes included in the traverser's logical view are visible to the traverser, and that nodes that are filtered out are invisible. The traverser's logical view of the data is determined when it is created. We will explain the creation process shortly.

A NodeIterator presents a flattened view of the tree as an ordered sequence of nodes, in document order. It has methods for moving forward and backward from the current node, but not up to its parent or down to its children. A TreeWalker preserves the hierarchical view of the subtree, allowing navigation in all directions. Specifically, this means that NodeIterators have only two navigation methods, whereas TreeWalkers have seven:

```
NodeIterator.nextNode()
NodeIterator.previousNode()

TreeWalker.nextNode()
TreeWalker.previousNode()
TreeWalker.firstChild()
TreeWalker.lastChild()
TreeWalker.nextSibling()
TreeWalker.previousSibling()
TreeWalker.parentNode()
```

In addition, TreeWalkers have access to the current node: `getCurrentNode()` simply returns the node without changing the TreeWalker's position, and `setCurrentNode()` allows the TreeWalker to jump to an arbitrary position in the tree.

Navigation Basics

We think of a traverser as always positioned either before the current node or after the last node in its logical view. When the traverser is created, it is positioned before the first node. The `nextNode()` method returns the current node and

advances the traverser. When the traversal is over, the method returns Null. A common pattern of working with a traverser is a While loop, as seen here:

```
// create a traverser called iter
    Node node;
    while (node = iter.nextNode()) doSomething(node);
```

The `previousNode()` method works the opposite way: it returns Null if the traverser is before the first node, and otherwise returns the node preceding the traverser and moves the traverser backward over the returned node.

Both kinds of traversers are "live": if, in processing a node, you modify a part of the tree that is visible to the traverser, then the traverser's logical view is immediately updated.

Creating a NodeIterator

The procedures for creating NodeIterators and TreeWalkers are identical, and both use a method of the DocumentTraversal interface. In DOMs that implement the Traversal module, the Document object also implements DocumentTraversal. That interface defines two "creation" methods: one for node iterators, and the other for tree-walkers. To attach a NodeIterator to a tree, convert (cast) the Document object to the TreeTraversal type and apply the `createNodeIterator()` method, giving it the root of the tree you want to traverse as the first argument:

```
    NodeIterator iter=
     ((DocumentTraversal)doc).createNodeIterator(
          root, NodeFilter.SHOW_ELEMENT, null, false);
    Node node;
    while (node = iter.nextNode()) doSomething(node);
```

This assumes that `doc` is the Document object and that `root` is a node. The While loop will visit, in document order, every Element node of the subtree under the `root`. It will skip nodes that are not elements. This is an example of how node iterators and tree-walkers present a filtered logical view of the data.

Creation methods take four arguments. The first of them, as we said, is a node, the root of the subtree to traverse. The last one is a Boolean that tells the traverser whether it should expand entity references if it comes across them. (It will be False in all our examples.) The second and third argument determine what is visible in the logical view presented by the traverser.

Filtering Parameters of createNodeIterator()

The second parameter of the two creation methods is a named integer constant. The names of the constants are formed by concatenating the string "SHOW_" with a node type. Thirteen such constants are defined in the NodeFilter interface: one for each of the twelve node types (listed in Table 4-2) plus SHOW_ALL. You can combine the constants using the "|" operator: to make only elements and attributes visible in the logical view, you would create an iterator like this:

```
NodeIterator iter=
  ((DocumentTraversal)doc).createNodeIterator(
       root, NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_ATTRIBUTE, null, false);
```

The third parameter of the two creation methods gives you an even finer control of what is included in the logical view. It is an object that, if not Null, must implement the `acceptNode()` method. The method takes a node as an argument and returns one of three possible integer values: FILTER_ACCEPT, FILTER_REJECT, or FILTER_SKIP. If the value is FILTER_ACCEPT, the node is visible in the logical view. If the value is NODE_SKIP, the node is skipped. If the value is NODE_REJECT, then a TreeWalker object will skip not only the current node but also all its children; for NodeIterator objects, SKIP and REJECT mean the same thing.

The `acceptNode()` method knows nothing about trees and traversals: it just takes each current node in turn and returns its verdict. This makes NodeFilter objects very easy to reuse: you can accumulate libraries of commonly used ones and plug them in as needed.

> **NOTE** *NodeFilter objects get to work after the integer-constants filters have already been applied. If the second argument says SHOW_TEXT and nothing else, the node filters will see only text nodes. It is your responsibility as a programmer to make sure that integer constants and node filters work harmoniously together.*

Now that we know how to create a traverser and how to navigate it, let us work through an example. The example will address a small problem with our add-address programs that we have been careful not to mention until now.

Example: The Problem of Odd Addresses

As shown in Listing 4-29, 4-30, and 4-31, all our add-address programs will produce somewhat unexpected results. Listing 4-32 shows a very simple page with tree addresses added.

Listing 4-32. Odd Addresses

```
<top attr="val" treeAddr="">
  <middle treeAddr="1"> middle-level content </middle>
  <middle treeAddr="3"> more middle-level content </middle>
  <middle treeAddr="5">
     <bottom treeAddr="5.1">bottom-level content.</bottom>
     <bottom treeAddr="5.3">more bottom-level content.</bottom>
  </middle>
</top>
```

Mysteriously, all the addresses are odd numbered, and there seem to be twice as many nodes as the eye can see. The expected result would be as shown in Listing 4-33.

Listing 4-33. Correct Addresses

```
<top attr="val" treeAddr="">
  <middle treeAddr="0"> middle-level content </middle>
  <middle treeAddr="1"> more middle-level content </middle>
  <middle treeAddr="2">
    <bottom treeAddr="2.0">bottom-level content.</bottom>
    <bottom treeAddr="2.1">more bottom-level content.</bottom>
  </middle>
</top>
```

The reason for this is that there are indeed invisible nodes, corresponding to the whitespace between the visible ones. If our documents had DTDs specifying that the content model of, for example, the top element is “children-only”, then the whitespace would not be preserved in the DOM tree. As it is, the parser has to play it safe and create nodes for them. To eliminate those whitespace-only nodes, we have to run another traversal. For this traversal, we will use a tree-walker, as shown in Listing 4-34.

Listing 4-34. Remove Whitespace-Only Nodes

```
public void removeWSNodes(Document doc) throws Exception {
  org.w3c.dom.traversal.TreeWalker tw
    = ((org.w3c.dom.traversal.DocumentTraversal)doc)
         .createTreeWalker(doc.getDocumentElement(),
                           org.w3c.dom.traversal.NodeFilter.SHOW_TEXT,
                           null,false);
  Node prev=tw.nextNode(); // node to be removed, if whitespace only
  Node node;
  while(null!=(node=tw.nextNode())){
    if(prev.getNodeType()!=org.w3c.dom.Node.TEXT_NODE)
      throw new Exception("non-Text Node from SHOW_TEXT TreeWalker");
    // trim whitespace from the text of the node
    String val=prev.getNodeValue().trim()
    if(val.length()==0) // there's nothing left!
     prev.getParentNode().removeChild(prev);
    prev=node;
  }
}
```

To obtain correct results, we could add this definition to our add-address pages, and also a new line of code (shown highlighted), right after the document is created and before the add-address procedure is invoked:

```
Document doc=db.parse(
                 new org.xml.sax.InputSource(
                    new java.io.StringReader(docStr)));
 removeWSNodes(doc); // the new line of code
 addTreeAddr(doc);
```

This would fix the problem with addTreeAddrCore.jsp and addTreeAddrRec.jsp.

Book Picker as DOM Builder

Our final example in this chapter will combine SAX and DOM to implement the following idea: suppose that you have a document that is too big to hold in memory all at once, and you need to process only one of its subtrees, anyway. However, the processing of that subtree is too complex and context-dependent to be handled by SAX combined with a simple state machine. (In our pickbook.jsp example, we had three states, but what if you need seventeen?) A natural strategy in this common situation is to use SAX to extract the subtree you want and build

a DOM tree for it. We will implement this idea as a revised book picker that extracts a specified book of the Bible and builds its DOM tree. For diversity, this program is implemented in Visual Basic, but the code is readily translatable into Java or, indeed, C#.

Running the Application

To run this (Windows-only) application, the file ActiveSAXbookpicker.dll must be installed on your machine and your IIS/PWS server must be running. (The DLL can be in any program directory; common choices are C:\Windows or C:\WinNT.) Assuming that ActiveSAXbookpicker.asp is in wwwroot\xmlp, the application is invoked by connecting to `http://localhost/xmlp/ActiveSAXbookpicker.asp`, as shown in Figure 4-8.

Figure 4-8. VB Book Picker

The code of `ActiveSAXbookpicker.asp` creates a DOMBuilder object defined in the ActiveSAXbookpicker.dll file and calls its `parseURL` method. The method takes two arguments: the URL of the biblical text and the name of the book to display. (Our code uses the same copy of ot.xml as in JSP applications; this assumes that Tomcat is running on port 8080.) The method returns an object of type `MSXML2.DOMDocument`. The object has an `xml` property which is a string that contains XML text. When the property is accessed, in the last line of code, the tree is serialized and the result is sent to the browser, as seen in Listing 4-35.

Listing 4-35. The ASP Page, ActiveSAXbookpicker.asp

```
<%@ LANGUAGE="VBSCRIPT"
%><% Option Explicit
 Dim dombuilder,doc,bookURI,bookName
 bookURI="http://localhost:8080/xmlp/dat/jb/ot.xml"
 ' This assumes that Tomcat is running,
 ' and webapps/xmlp/dat/jb/ot.xml is in place,
 ' but if not, simply copy ot.xml to wwwroot\xmlp\ot.xml
 ' and tstmt.dtd to wwwroot\common\tstmt.dtd
 ' set bookURI="http://localhost/xmlp/ot.xml"
 bookName="JOB"
 ' bookName can, of course, be pulled out of QueryString object
 Set dombuilder = CreateObject("ActiveSAXbookpicker.DOMBuilder")
 Set doc = dombuilder.parseURL(CStr(bookURI),CStr(bookName))
Response.ContentType = "text/xml"
 Response.write(doc.xml)
%>
```

This ASP page invokes an ActiveX control, MSXML, which in turn tries to access a URL resource. For this to work, you need a specific combination of security settings on the Internet Explorer Tools ➢ Internet Options ➢ Security menu. This combination is different on Windows 9x and on Windows 2000. In particular, on Windows 2000, you may need to put `http://localhost` and `http://localhost:8080`, or even the specific URL of the pages you are downloading onto Trusted Sites. Future versions of the browser and/or the operating system may require additional security setting modifications.

The Application's Code

The code of the application consists of three files: DOMBuilder.cls, which defines the top-level `parseURL()` function; ContentHandlerImpl.cls, which does SAX parsing and builds a DOM tree from SAX output; and ErrorHandlerImpl.cls, which processes errors (and also returns a DOM tree that holds the XML error message). We will discuss these in that order.

The DOMBuilder Class

The DOMBuilder class defines a single `parseURL()` function that takes two arguments—a URL and a string—and returns a DOM object. It is preceded by fourteen lines of framework-generated code that sets the class properties. The function itself instantiates a SAX parser (SAXXMLReader), sets its handlers to our application-specific ContentHandlerImpl and ErrorHandlerImpl, and calls its own `parseURL()` function that takes a single argument, a URL. (See Listing 4-36.)

Listing 4-36. The DOMBuilder's `ParseURL()` *Function*

```
Public Function parseURL(srcURL As String, matchStr As String)
  As MSXML2.DOMDocument ' returns a MSXML2.DOMDocument
Dim reader As New SAXXMLReader ' a SAX parser
Dim contentHandler As New ContentHandlerImpl ' Receives parsing events
Dim errorHandler As New ErrorHandlerImpl ' Receives error events
Set reader.contentHandler = contentHandler
Set reader.errorHandler = errorHandler
' set the public property titleString to matchStr (calls Property Let)
contentHandler.titleString = matchStr
On Error GoTo Parse_Err ' go to Parse_Err: label
reader.parseURL (srcURL) ' parse the input

' return value from function via function name
Set parseURL = contentHandler.doc()
Exit Function

Parse_Err:
' return value from function via function name
Set parseURL = errorHandler.doc()
End Function
```

The ContentHandlerImpl Class

The ContentHandlerImpl class is quite similar to our Java BookPicker class except that it also builds a DOM object. You may want to review Listing 4-14 from earlier in the chapter to see the overall logic of the program. In particular, recall that the content handler can be in one of three states: the silent state that produces no output, the echo state that copies the input to output because we are in the right book, and the in-title state when we don't yet know whether we are in the right book or not and must accumulate the entire title before switching into either the echo or the silent state.

The code of ContentHandlerImpl consists of four sections:

- Framework-generated prefatory matter
- Variables and properties, declared and instantiated
- Overridden SAX callbacks: startDocument, startElement, endElement, and characters
- Framework-generated empty stubs for the remaining callbacks

We will show the two middle sections. The first of them requires little comment, except to remind you that the content handler can be in one of three states. (See Listing 4-37).

Listing 4-37. Enumeration, Variables, and Properties

```
Implements IVBSAXContentHandler ' the Implements clause
' an enumeration of SAX states
Private Enum saxState
   saxSilent = 0
   saxInBkTLong = 1
   saxEcho = 2
End Enum

'private object variables (DOM Document and Node)
Private oDoc As MSXML2.DOMDocument
Private oNode As MSXML2.IXMLDOMNode

' three non-object variables
Dim sState As saxState
Dim bkTitle As String
Dim strMatch As String

' access to private variables: Property Get and Property Let
Public Property Get doc() As MSXML2.DOMDocument
 Set doc = oDoc
End Property
Public Property Let titleString(str As String)
 strMatch = str
End Property
```

SAX Callbacks

The biggest difference between the Java version and this version is that we provide for the possibility that there may be more than one book title matching the user input and, therefore, that more than one book may be sent to output. In the earlier version, this would result in non-well-formed XML consisting of more than one tree. In this version, we provide a top wrapper element called `books`. It is created in `startDocument()` using DOM's `createElement()` and `appendChild()` methods. (See Listing 4-38.)

Listing 4-38. `startDocument()`

```
Private Sub IVBSAXContentHandler_startDocument()
Set oDoc = New MSXML2.DOMDocument
Call oDoc.appendChild(oDoc.createElement("books"))
Set oNode = oDoc.documentElement
sState = saxSilent
End Sub
```

In `startElement()`, the program can be in only one of two possible states: silent or echo. If it is in the echo state, we copy the element to the emerging DOM tree, again using DOM's `createElement()` and `appendChild()` methods. If we are in the silent state, we check to see whether the element happens to be the book title element. If so, we initialize the `bkTitle` string:

```
Private Sub IVBSAXContentHandler_startElement(
  strNamespaceURI As String,
  strLocalName As String,
  strQName As String,
  ByVal oAttributes As MSXML2.IVBSAXAttributes
)
If sState = saxEcho Then
Set oNode = oNode.appendChild(oDoc.createElement(strLocalName))
ElseIf sState = saxSilent Then
   If strLocalName = "bktlong" Then
      sState = saxInBkTLong
      bkTitle = ""
   End If
End If
End Sub
```

In `endElement()`, we can be in any of the three states. If we are in the silent state, we do nothing. If we are in the echo state, we check to see whether we have reached the end of the book we've been echoing; if so, we reset the state to silent. Finally, if we are in the in-title state, we want to check whether the title that we have just finished accumulating matches the `strMatch` argument submitted by the user. If it does, we switch to the echo state; if it doesn't, we switch to the silent state:

```
Private Sub IVBSAXContentHandler_endElement(
  strNamespaceURI As String,
  strLocalName As String,
  strQName As String
)
```

```
If sState = saxEcho Then
  Set oNode = oNode.parentNode
  If strLocalName = "book" Then sState = saxSilent
ElseIf sState = saxInBkTLong Then
  If (InStr(bkTitle, strMatch)) Then
    sState = saxEcho
    ' must build book node, then bktlong within it
    Dim oTitle As IXMLDOMElement
    Set oTitle = oDoc.createElement("bktlong")
    Call oTitle.appendChild(oDoc.createTextNode(bkTitle))
    Dim oBook As IXMLDOMElement
    Set oBook = oDoc.createElement("book")
    Call oBook.appendChild(oTitle)
    Call oNode.appendChild(oBook)
    Set oNode = oBook
    bkTitle = ""
  Else ' was in bktlong, but title didn't match
    sState = saxSilent
    bkTitle = ""
  End If
End If
End Sub
```

The `characters()` method does nothing if we are in the silent state, accumulates the title if we are in the in-title state, and copies to the emergent DOM tree if we are in the echo state. Because we are copying text, we copy into a text node rather than an element node:

```
Private Sub IVBSAXContentHandler_characters(strChars As String)
If sState = saxEcho Then
  Call oNode.appendChild(oDoc.createTextNode(strChars))
ElseIf sState = saxInBkTLong Then
  bkTitle = bkTitle & strChars
End If
End Sub
```

The rest of the code consists of the framework-generated stubs for the remaining callbacks, such as

```
Private Sub IVBSAXContentHandler_endDocument()
End Sub
```

This concludes the `ContentHandlerImpl` class. The `ErrorHandlerImpl` has no features, DOM or SAX, that are not in `ContentHandlerImpl`.

Conclusion

In this chapter, we have covered SAX and DOM programming, from the basics to more advanced topics, separately and in combination. We have also introduced more JAXP material, including the Transformer object for XSLT transformations and XML data conversions among different representations.

Unlike every other specification in the book, SAX is not sponsored by W3C, OASIS, or any other organization. It is a de facto standard, designed and implemented by a group of developers on the xml-dev mailing list (with David Megginson as the focal point and implementer in charge). Its APIs are exposed by all major parsers, including MSXML. However, they are not supported by the .NET XML classes; in particular, the XmlReader class of the .NET framework has nothing to do with the SAX XMLReader interface. In other words, XML processing in .NET is different from MSXML, and, if you want to use SAX interfaces, you will have to import MSXML as a separate package or implement SAX on top of .NET's XmlReader.

We also covered DOM interfaces and developed an example of DOM and SAX used together and implemented in VB/ASP. This example is easy to generalize to many situations in which you want to extract a subtree from your XML data before building a DOM.

CHAPTER 5

XPath, XSLT, and XLink Processing

In this chapter and the next, we explore two languages—XPath and XSLT—that were first introduced in Chapter 1. Our aim is to present an overall picture: the data model, the syntax, the processing model, and essential programming techniques. We will not try to be complete and exhaustive: this has been done, in a definitive way, in Kay's *XSLT Programmer's Reference* (a book twice the size of this one and completely dedicated to XSLT). We will concentrate on those parts of XSLT that are most commonly used, but, within that scope, we will go into considerable depth on issues that are important to programmers: idiomatic XSLT programming, design patterns, and efficiency. In general, we try to treat XSLT as just another programming language.

The XML material used in this chapter's XSLT programs often contains extended XLink structures. Together, the programs form a complete application that builds a collection of XLinks from user input and performs search operations on the resulting graph structure. You may want to review the two sections of Chapter 2 that presented XLink and a simple XLink application ("XLink Attributes and XLink Graphs" and "An XLink Example"). Parts of that material are reused in this chapter.

In outline, the chapter proceeds as follows:

- an XLink application: creating and using a linkbase
- XPath data model and expression syntax
- XSLT overview and processing model
- the push programming pattern (data-driven control structure, default templates)
- subroutines and recursion
- the code of the linkbase application

An XLink Application: Creating and Using a Linkbase

Our XLink application consists of two parts. Part 1 creates a linkbase from an XML "link source" file that itself can be easily created from user input via an HTML form. The linkbase contains extended link structures that describe many-to-many links in XML data that can be in a single document or multiple documents. Part 2 implements two sample queries.

We again use the King James Bible (ot.xml and nt.xml) as our XML data. As you may know, New Testament texts frequently quote or contain parallels to the Old Testament, and there are many parallels among the four Gospels. Our links are based on those parallels and quotes, as documented in *The New Oxford Annotated Bible with the Apocrypha/Deuterocanonical Books* (ISBN 0-19-528485-2).

The screenshot in Figure 5-1 shows the result of a query.

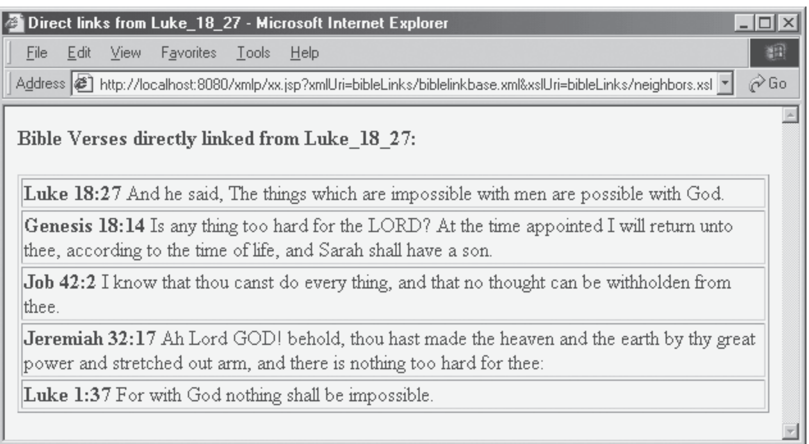

Figure 5-1. Linkbase query result, simple query

From Link Source to Linkbase

Our link source file is bibleLinks/biblelinks.xml. It consists of `ref` elements as shown in Listing 5-1.

Listing 5-1. Link Source Document

```
<bible-linkbase-src>
 <ref>
   <from>nt.xml Luke 18 27</from>
   <to>ot.xml Genesis 18 14</to>
 </ref>
 <ref>
<!-- more ref elements -->
</bible-linkbase-src>
```

Each `ref` element establishes a one-way relationship between two items. To express this information in an extended link, we need two `locator` elements for the two items, and an arc element to assert the relationship. (See Chapter 2 for more detail.) The entire linkbase document will consist of a single extended link element that contains all the `locator` and `arc` elements. The sample of Listing 5-1 should come out as shown in Listing 5-2.

Listing 5-2. Linkbase Document

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE elt PUBLIC "elt"
      "http://localhost:8080/xmlp/bibleLinks/biblelinkbase.dtd">
<elt xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
   <elt xlink:label="Luke_18_27"
        xlink:type="locator"
        xlink:href= // long URL, including an XPointer, broken over two lines
      // see explanations to Listing 2-7 in Chapter 2
      "dat/jb/nt.xml#
      xpointer(/tstmt/bookcoll/book[bktshort='Luke']/chapter[18]/v[27])"
   />
   <elt xlink:label="Genesis_18_14"
        xlink:type="locator"
        xlink:href= // long URL, including an XPointer, broken over two lines
        "dat/jb/ot.xml#
        xpointer(/tstmt/bookcoll/book[bktshort='Genesis']/chapter[18]/v[14])"
   />
   <!-- more locator elements -->
   <elt
 xlink:to="Genesis_18_14"
 xlink:from="Luke_18_27"
 xlink:type="arc"/>
<!-- more arc elements -->
</elt>
```

The transformation is performed by ref2link.xsl. We are not yet ready to read its code, but let us list some of the tasks it has to perform:

- Process each `ref` element twice—first to construct locators, then to construct arcs.
- In constructing locator labels, convert a string like "nt.xml Luke 18 27" into the label "Luke_18_27". (The same labels are used in `arc` elements.)
- In constructing locator `href` attributes, convert a string like "nt.xml Luke 18 27" into a complete URL+XPointer expression.
- For either labels or `href` attributes, one of the tasks is to break the original string into pieces using the space character as the separator.

Once the linkbase is built, we can run queries. We implement two of them:

- Given an element, find all elements to which it is linked.
- Given an element, find all elements that are reachable from a given element by recursively going from its neighbors to its neighbors' neighbors, and so on until no new elements are found.

In technical terms, the second query is the *transitive closure* of the first. To give a familiar example of the same concept, in a tree structure, *ancestor* is the transitive closure of *parent*. By the end of this chapter, after a heavy dose of XPath and XSLT, you will be able to read the code of the queries and improve it in exercises.

The XPath Language and Data Model

XPath is an expression-oriented language: its main syntactic unit is an *expression* that gets evaluated to produce a *value*. To learn XPath is to learn three things:

- What kinds of values are there? (the data model)
- What kinds of expressions are there? (the syntax)
- How are expressions evaluated? (the evaluation function from expressions to values)

Before we work our way through these questions, we would like to give you a program—an expression evaluator—that you could use to try out various

expressions. To use such a program, you will need a little background in XPath data types and the evaluation process. So we will work through a short example first, present the evaluator, and then present the language in more detail.

XPath is always used within another application, such as XSLT, XPointer, or (currently vendor-specific) functions that retrieve node-sets from DOM trees using XPath. The outside application receives the value of an XPath expression and does something with it. The outside application also provides a context of evaluation for XPath expressions. In this chapter, we are interested in how XPath is used in XSLT; all our examples of XPath are XSLT examples.

XPath Data Types and the Context of Evaluation

The four data types in XPath are string, number, Boolean, and node-set. The first three are familiar, and we will refer to them collectively as scalar data types. A node-set, as the name suggests, is a set of nodes in a tree. Expressions corresponding to *scalar data types* use the same syntax in XPath as in other programming languages. Path expressions that evaluate to node-sets are the core of the language. To illustrate path expressions and node-sets, consider a simple stylesheet, similar to Listing 1-13 of Chapter 1 but without variables. It is intended for the same pdata.xml data that we have used in Chapters 1 through 3. The output of the program is an HTML table that lists the title and first and last names of each person, as shown in Listing 5-3.

Listing 5-3. XPath Within XSLT (pdataNameTable.xsl)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
<xsl:template match="/" >
  <html><head><title>People's names</title></head>
    <body><table border="1">
     <xsl:for-each select="pdata/person/name">
      <tr>
 <td><xsl:value-of select="title"/></td>
 <td><xsl:value-of select="first"/></td>
 <td><xsl:value-of select="last"/></td>
      </tr>
     </xsl:for-each>
   </table></body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The XPath expressions in this program are the values of the `match` and `select` attributes. They are all path expressions. The first of them, the value of the `match` attribute in line 3, matches a node-set that consists of a single node, the root of

the tree. It creates a context in which the contents of the template element are evaluated.

The next path expression is the value of the `select` attribute in the `xsl:for-each` element. It is a relative path expression (it does not start with a "`/`"), and it is evaluated in the context created by the match attribute. The value of this expression is a node-set that consists of all the `/pdata/person/name` elements; if the file has seventeen person records, the node-set will contain seventeen nodes. The contents of the `xsl:for-each` element are applied to each of those nodes in turn. Each node, in turn, becomes the current node that forms (a part of) the context in which the contents of the `xsl:for-each` element are evaluated.

More precisely, the context of evaluation consists of the following:

- a context node-set and a specific node (the context node) within it
- two positive integers: the context node position within the context node-set and the context size (that is, the number of nodes in the node-set)
- a set of variable bindings (XPath can refer to variables but cannot create them: they are created in the application that uses XPath.)
- a set of namespace declarations that are in scope for the expression being evaluated
- a function library (XPath has a core function library that can be extended with additional functions by the application that uses XPath.)

XPath Examples

Here are a few XPath expressions, with explanations. When talking about values, we use boldface to indicate that we are talking about objects rather than expressions that refer to those objects. For instance, `70` is a number expression, but **70** is a number object.

When Boolean examples appear in an XSLT stylesheet (or any other XML document) the < and > characters are, of course, replaced by the appropriate entities, according to XML syntax. Remember that, before an XSLT program gets to the XSLT processor, it first goes through an XML parser.

- `'abcd'` is a string literal whose value is the string **abcd**.
- `concat('ab','cd')` is a functional expression whose value is the same string.
- `325.74` is a number expression whose value is the number **325.74**.

- `position()` is a functional expression whose value is a number, the position of the context node within the context node-set. (The position of the first node is 1.)

- `5>3` is a Boolean expression whose value is the Boolean value true.

- position()<5 and `position()>2` is another Boolean expression.

- `$var>17` is another Boolean expression whose value is **true** if the value of the variable `var` (interpreted as number, possibly after conversion) is greater than **17**.

The nature of context components and where they come from will become clear as we proceed. The main thing to remember is that the context of evaluation is external to XPath: it is supplied by the application that uses it. The role of XPath is to select a set of nodes for processing by its mother application.

A JSP for Testing

We provide a Web application for evaluating (that is, computing the value of) XPath expressions. The application is xptrans.jsp. To evaluate an XPath expression, our evaluator constructs an XSLT stylesheet and runs it, from within the program, on the specified XML source. We also show how the same kind of evaluator can be implemented in ASP.

We present the evaluator here so you can start using it, but many details of its code require some knowledge of XSLT and XPath. You don't have to work through the code to use the application: simply complete and submit the form in the application's entry page, moreXSL\xptrans.htm.

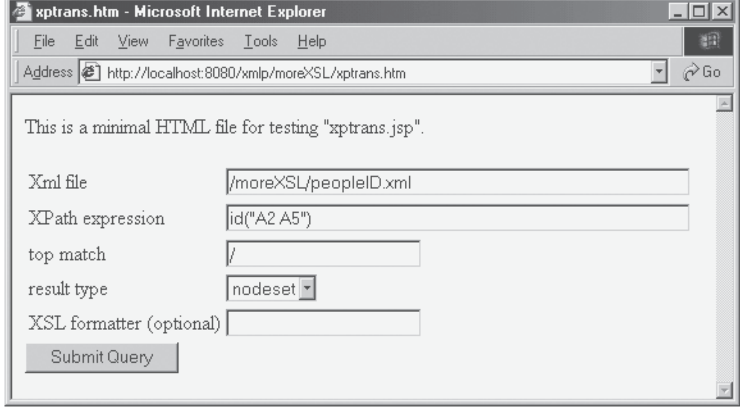

Figure 5-2. The entry form for XPath tester

As you can see from the screenshot, the program expects four parameters from the reader:

- *XML URI*: an XML source to be processed by the stylesheet. (It can be a URL or a local file.)

- *top match*: the value for the match attribute in the top-level template.

- *XPath expression*: the path expression to be evaluated in the context created by the top match.

- *result type*: `nodeset` and `value`. Use `nodeset` if the expected result is either a single node or a set of nodes. Use `value` if the expected result is a string, a number, or a Boolean. (If the result is a set of nodes and you ask for its value, you will get the string value of the first node.)

With the defaults as shown in Figure 5-2, the output is as follows:

```
<list>
<person helpers="A3 A4 A5" id="A2">
<name>
<title>Mr.</title>
<last>Bargle</last>
<first>Bertrand</first>
</name>
</person>
<person helpers="A1" id="A5">
...
</list>
```

You may want to start by changing only the XPath expression, until you learn more XPath and XSLT and can start experimenting with the interaction between the match variable that establishes the context of evaluation and the XPath expression evaluated in that context. To experiment with Boolean, number, and string expressions, set the result type to "value" and enter the XPath expression to evaluate. The result will appear in the browser window. (This will be useful as we go through XPath functions and operators in a later section.)

JSP (Java) Code

The code of xptrans.jsp, in outline, proceeds as follows:

1. Import needed libraries and classes in the opening page directive.

2. Create a Hashtable object called `cache` in a `useBean` action. This is only done once when the application is run for the first time, because the object has application scope. After that, the cache and its contents persist from one call to the next.

3. Retrieve user-submitted values for the five parameters from the Request object.

4. Retrieve a parsed Document (DOM) object for the given XML source from cache; if it is not there, parse and store the result in cache. Also store in cache the DocumentBuilderFactory object from which a parser instance can be obtained.

5. Construct an XSLT program as a text string using the values of `templateMatch` and `xPath`.

6. Construct a transformer object and run the XSLT program on the parsed document.

7. If `formatXslUri` is empty, write the result to the output stream; otherwise, construct another transformer object and run it on the output of the first transformer.

Many details of this program are quite similar to what we did in xpn.jsp within the XLink application of Chapter 2. The xpn.jsp file retrieves content from an XML document (ot.xml) using an XPath expression. You may want to review that section before proceeding. The big difference from xpn.jsp is in the way in which the DOM object is created. In Chapter 2, for simplicity, we used Apache-specific code. Here, we use generic JAXP code as in Chapter 4.

The way that the parser is invoked to create a DOM object is quite different from the xpn.jsp version of Chapter 2, where we used Apache-specific code for the purpose. In this chapter, we use the generic JAXP (Java API for XML Processing). With JAXP, you can switch from one parser to another by simply changing a system property that points to the class that implements the DocumentBuilderFactory interface. Here is the code; the two highlighted lines set the parser properties:

```
  if(0>xmlUri.indexOf(":")) xmlUri="file:///"+application.getRealPath(xmlUri);
 Document doc=(Document) cache.get(xmlUri); // attempt to retrieve from cache
  if(null==doc){
    DocumentBuilderFactory dbf=(DocumentBuilderFactory) cache.get("dbf");
    if(null==dbf){
```

```
      dbf=DocumentBuilderFactory.newInstance();
      dbf.setNamespaceAware(true); // be namespace aware
      dbf.setValidating(false); // do not validate
      cache.put("dbf",dbf);
    }
    DocumentBuilder db=dbf.newDocumentBuilder();
    doc=db.parse(xmlUri);
    cache.put(xmlUri,doc);
  }
```

In ASP, the same result (minus caching) can be obtained by the following code:

```
Dim objXML,xmlFile
xmlUri =QueryString.item("x")
Set objXML = Server.CreateObject("MSXML2.DOMDocument.3.0")
objXML.load Server.MapPath(xmlUri) ' load parses the file
```

The next step is to construct the text of the stylesheet by repeatedly concatenating all the pieces into a single string variable, `xslString`. For the given defaults, the resulting stylesheet looks as follows (the values of the `match` and `select` attributes have been inserted from user input):

```
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
<xsl:output method='xml'/>
<xsl:template match='/'>
<list><xsl:copy-of select='*'/></list>
</xsl:template>
</xsl:stylesheet>
```

If the value of templateMatch is not “/”, then there is a possibility that XSLT’s default template matching rules will generate extraneous output. To suppress that output, we override the default rules with a one-line template (also constructed) that has an empty template body and therefore produces no output:

```
<xsl:template match='text()'/>
```

At this point in the code, we are ready to run the stylesheet on the DOM object. As in the preceding chapter, we create a Transformer object and supply it with a DOM source and a stream result. However, unlike as in Chapter 4, our transformer is created with a nonempty stylesheet and therefore does more than simply serializes the DOM source to an XML text: it performs a real transformation as specified in `xslString`. In effect, we are running XSLT code from within

a Java program. Similar facilities exist for VBScript, JScript, VB, Python, and other languages. In Java, we use JAXP interfaces to construct an XSLT processor object and run the stylesheet:

```
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer =
  tFactory.newTransformer(new StreamSource(new StringReader(xslString)));
for(Enumeration e = props.propertyNames(); e.hasMoreElements();){
  String name=(String)e.nextElement();
  String val=props.getProperty(name);
  transformer.setParameter(name,val);
  }
StringWriter outStrW=new StringWriter();
response.setContentType("text/xml");
transformer.transform(new DOMSource(doc), new StreamResult(outStrW));
out.write(outStrW.toString());
```

The last two lines save the output of transformation in a StringWriter object (which is just a string with stream interfaces) and write the object to the output stream. We do that for ease of debugging, so the output can be inspected inside the program before it pops up in the browser. In ASP, we would send the result directly to the output stream:

```
xmlObj.transformNodeToObject(objXSL,Response)
```

NOTE *We avoid using strings when working with MSXML because they are always, by definition, stored in UTF-16. MSXML can output byte sequences in other encodings, but string storage implies a coercion to UTF-16. Trying to set the charset to something else (for example, for compatibility with a receiving application that doesn't understand UTF-16) simply won't work. At best, you will get a comprehensible error message. Microsoft recommends that output be sent directly to the ASP response stream, as in* `xmlObj.transformNodeToObject(xslObj, Response)`.

Running XSLT from other code is a common way of integrating XSLT into larger systems. Conversely, functions written in general-purpose programming languages can be used from within XSLT. We will have much more to say about interactions between XSLT and other programming languages in the next chapter.

XPath Values and Data Types

As previously stated, XPath has four data types: Boolean, number, string, and node-set.

The Boolean data type has just two possible values: True and False.

The number data type of XPath is technically known as "the IEEE double-precision 64-bit floating point type [IEEE 754-1985]." It is identical to `double` in Java and C#. We will say more about it in Chapter 9 when discussing the data type library of XML Schema Part 2.

The string data type is a sequence of Unicode/UCS characters. (*UCS* stands for *Universal Character Set*, an ISO standard for character representation that corresponds to Unicode.) XPath characters are the same as *XML 1.0* characters.

The most important data type is node-set, an unordered set of nodes without repetition, the result of evaluating of a path expression. XPath node-sets are unstructured, but the nodes they contain come from an XPath tree. Although node-sets are unordered, they are frequently processed in document order, which, for elements, is the order of start tags in the serialized document. We discuss the node-set data type and the XPath tree model in a separate section.

Type Conversions

For each data type except node-set, there is a conversion function of the same name: `boolean()`, `number()`, and `string()`. In many contexts, conversion occurs automatically, and automatic conversion always produces the same result as the conversion function.

Conversions between Booleans, numbers, and strings work the way you would expect: the string "false" is converted to the Boolean value False, and the number 17 is converted to the string "17". A node-set is converted to a string by returning the string value of a single node: the node that is the first in document order. If the node-set is empty, an empty string is returned. A node-set is converted to a number by first converting it to a string, then applying string-to-number conversion. A node-set is converted to Boolean False if it is empty; otherwise, it is converted to True.

XPath 1.0 does not have a function to convert to node-set. Within XSLT, there is an additional data type called *result tree fragment* that is quite awkward to work with unless it is converted to node-set. All major XSLT processors now have a `node-set()` function that converts result tree fragments to node-sets, and you will see it in our code. *XPath 2.0* is expected to remove the result tree fragment data type altogether, eliminating the need for this function.

Expressions Other Than Path Expressions

The syntax of Boolean, number, and string expressions does not require much of an explanation. As in many languages, there are primary expressions (including function call expressions), a set of functions to learn, and operators that combine primary expressions into compound ones. Here are the primary expressions:

- *VariableReference*: $book (the variable name is "book")
- *Parenthetical expression*: ($price*0.8 + $shipping)
- *Quoted String Literal*: "Hello, XPath", 'single quotes are "ok" too'
- *Number Literal*: 325, 4.76E25
- *Function Call*: count()

Functions and Function Libraries

Function calls use a function from the function library that is part of the evaluation context: a function library is a mapping from function names to functions. XPath itself contains a core function library; both XSLT and XPointer extend it with additional functions. The core library contains Boolean, number, string, and node-set functions. Here is a selection of the more commonly used Boolean, number, and string functions, showing their signatures (node-set functions will be presented in the next section):

```
boolean not(boolean)

number floor(number)
number ceiling(number)
number round(number)

string concat(string, string, . . . )
boolean starts-with(string, string)
boolean contains(string, string)
string substring-before(string, string)
string substring-after(string, string)
string substring(string, number)
string substring(string, number, number)
number string-length(string)
```

These functions are mostly self-explanatory. You can find details of their use in Kay's book or the XPath recommendation, and you will see many of them used in our code. In the meantime, you can experiment with string, number, and Boolean functions using xptrans.jsp. Set the result type to "value" and enter an XPath functional expression to evaluate, such as round(3.14), not(5>3), substring-before("abcde","cd"), or contains("abcde","cd"). The result will appear in the browser window.

XPath Operators

Primary expressions can be combined using operators. XPath has no string operators, but there are four number operators: the familiar + - * for addition, subtraction, and multiplication, and div for division. (The "/" character is used for a different purpose in path expressions.)

The Boolean operators are as follows (grouped by precedence, lowest precedence first):

- or
- and
- =, !=
- <=, <, >=, >

When the "<" and "<=" operators appear in XSLT, the "<" character must be escaped according to XML rules (as in <xsl:if test="$var < 10 . . . >).

We will see node-set operators in the section on path expressions. In the meantime, you can experiment with string, number, and Boolean expressions using xptrans.jsp. Set the result type to "value" and enter an XPath expression to evaluate, such as 3 + 5 or 7 > 4. The result will appear in the browser window. Remember to escape the less-than operator as < (or don't, and view the error message).

XPath Data Model and the Node-Set Data Type

We will re-use Listing 1-2 of Chapter 1 (presented here as Listing 5-4) to illustrate the XPath data model. In Chapter 1, it was also used to illustrate XPath, but our diagram and our model are now more detailed.

Listing 5-4. An XML Document with Attributes and Namespaces

```
<?xml version="1.0"?>
<!-- This is a comment. The next line is a PI -->
<?xml-stylesheet href="exchange.css" type="text/css"?>
<exchange tone="informal">
  <q>What's up?</q>
  <a>Nothing much. (I am the root, you know.)</a>
</exchange>
```

The corresponding XPath tree is shown in Figure 5-3. We follow Michael Kay's convention of representing each node by a box divided into three parts: the top part shows the type of the node, the second gives its name (if any: the root, text, and comment nodes do not have names), and the third part shows its content, if any.

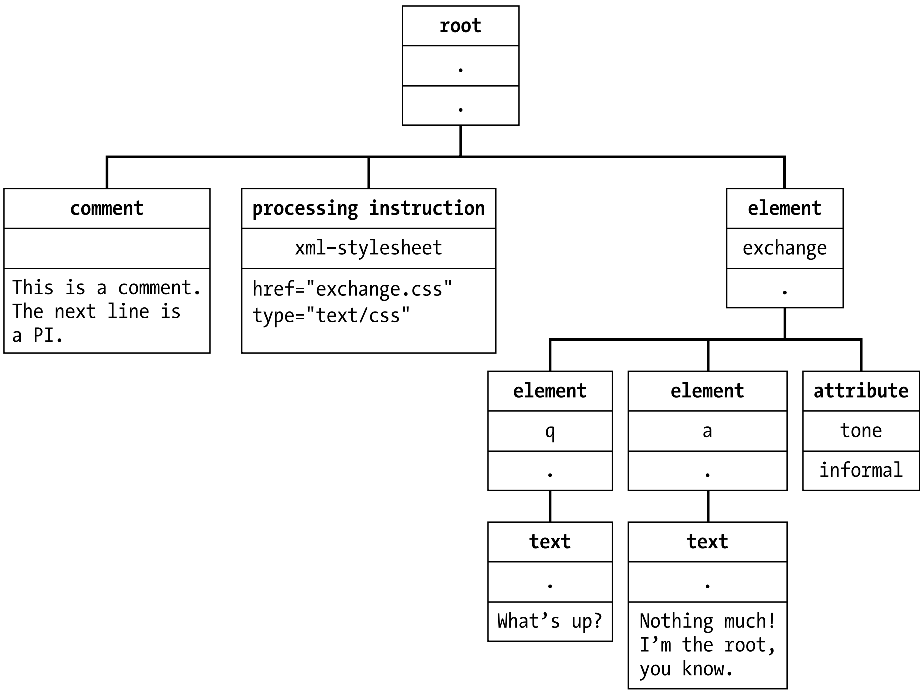

Figure 5-3. An XPath tree for Listing 5-4

Note again the following two conventions:

- There is a root node that is parent to the root of the element tree, as well as top-level comments and processing instructions.
- The text content of an element is wrapped in a text node.

These are reflected in the XPath system of node types.

Node Types and Node Properties

A node-set can contain nodes of different types. XPath has seven such types:

- the root node (parent of top-level comments, PIs, and the root element node)
- element nodes (form a tree of elements)
- text nodes (hold the text content of element nodes)
- comment nodes
- processing instruction nodes
- attribute nodes (along a separate axis)
- namespace nodes (along a separate axis)

These node types are not exactly equal: the first five are structural types that affect the shape of the tree whereas the last two are more like decorations on other nodes. (We call them decorative nodes.) XPath treats structural types and decorative types differently, as you will see in a moment.

Nodes have properties, such as name and text value, and different types of nodes have somewhat different sets of properties. Table 5-1 provides a summary.

Table 5-1. Node Types and Node Properties

NODE TYPES	NAME	TEXTVALUE	PARENT	CHILDREN
root		Concatenation of children's text values	No parent	element root, comments, PIs
element	Tag name	Concatenation of children's text values parent element	Elements	any but root
text		Own text value	Element node	No children
comment		Comment text	Root or containing element	No children
PI	First token	Remaining tokens	Root or containing element	No children
attribute	Attribute name	Attribute value	Owner element	No children
namespace	Namespace URI	Namespace URI	Owner element	No children

The Axes of the XPath Tree

XPath's main function is to select a set of nodes for processing by its mother application. The selection process (that is, the process of evaluating a path expression) always starts from the context node, and proceeds to navigate the tree guided by the path expression that is being evaluated. The navigation part can be empty, in which case the selected node-set consists of the context node itself. Otherwise, there are four directions to navigate: up or down or forward or back. (Remember that the tree is two-dimensional and arranged from left to right in document order.) In addition, nodes can have decorations (attributes and namespaces), and you may choose to select those. In the terminology of XPath, each of these possibilities is called an *axis*. We divide them into three groups:

- close-by axes, such as child, that include all nodes right below you

- more far-reaching axes, such as descendant, that include all the nodes underneath you all the way to the leaves (this is child applied recursively, or the transitive closure of child)

- two decorative axes, attribute and namespace, that contain the corresponding node types

Another useful way to look at XML data is to switch to the linear view and divide all elements into groups by the mutual position of their tags. The four possibilities are as follows:

- Ancestors of element E are all elements whose start tag precedes the start tag of E and whose end tag follows the end tag of E.

- Descendants of element E are all elements whose start tag follows the start tag of E and whose end tag precedes the end tag of E.

- Preceding elements of E are all elements whose end tag precedes the start tag of E.

- Following elements of E are all elements whose start tag follows the end tag of E.

XPath axes capture these distinctions. There are thirteen axes altogether:

- *self:* the context node itself
- *parent:* the parent of the context node (empty if the context node is the root)
- *child:* the children of the context node.
- *preceding-sibling:* all preceding (in document order) siblings of the context node
- *following-sibling:* all following (in document order) siblings of the context node
- *ancestor:* transitive closure of parent
- *descendant:* transitive closure of child
- *ancestor-or-self:* exactly what it says it is
- *descendant-or-self:* same
- *preceding:* as defined previously, all elements whose end tag precedes the start tag of the context node
- *following:* as defined previously, all elements whose start tag follows the end tag of the context node
- *attribute axis:* all attributes of the context node, if any. Empty for non-element nodes and elements without attributes.
- *namespace axis:* all namespace nodes of the context node, both defined and inherited from ancestor nodes.

If the context node is a structure node, then the union of self, ancestor, descendant, preceding, and following will contain all the element nodes of the tree. It will not contain the decorative (attribute and namespace) nodes. Each decorative node has a parent—the element node that owns them—but decorative nodes are not their parent's children. In other words, if you ask for an element node's children, you will get only its children elements, comments, and processing instructions, but not its attributes or its namespaces.

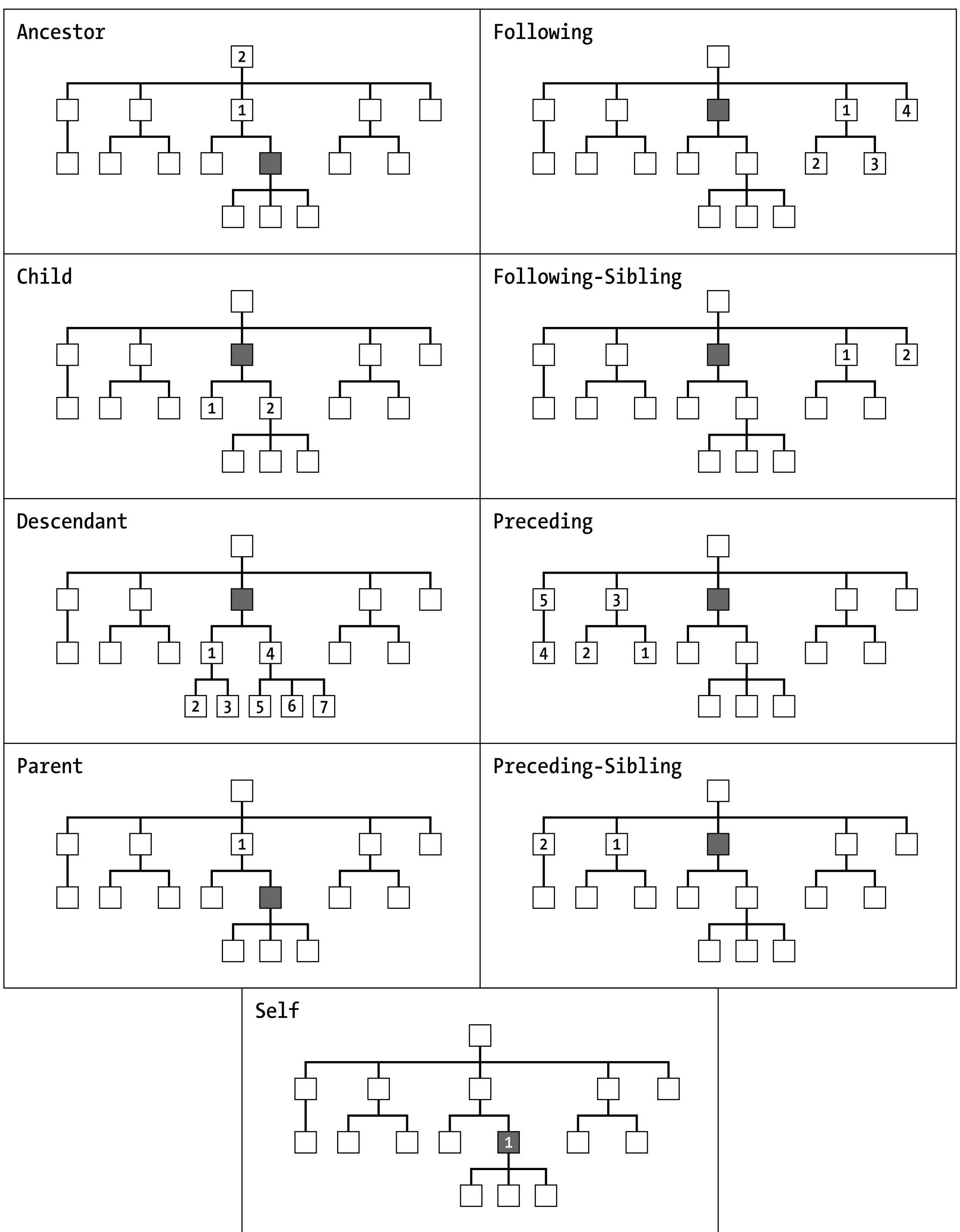

Figure 5-4. Positional axes

Axes and Context Node Position

XPath expressions are evaluated in a context that includes a context node and its position within the node-set. The position is determined differently depending on the axis. For forward axes (child, descendant, descendant-or-self, following-sibling, and following), node positions are numbered from the context node forward in document order; for reverse axes (parent, ancestor, ancestor-or-self, preceding-sibling, and preceding) node positions are numbered from the context node backward, in reverse document order. This and other details are shown in Figure 5-4. The numbers in the diagram show how the nodes are ordered along the axes. As you can see, the "document order" is the order of the depth-first traversal of the element tree: we go as deeply as we can into each subtree before visiting the next one.

Of the eleven positional axes, we show all except descendant-or-self and ancestor-or-self.

We are now ready to discuss path expressions and how they evaluate to node-sets.

Path Expressions

Path expressions (whose formal name is *location path expressions*) come in two forms: full and abbreviated. All examples of Chapter 1 used the abbreviated form because it is easier to read and requires less background. However, it is the full form that is primary; abbreviated form is derived from full form by a few simple rules. Only a subset of full-form expressions can be converted to the corresponding abbreviated form.

Both full-form and abbreviated-form expressions can be *absolute* or *relative*. Absolute expressions start with the "`/`" character, and relative don't. Relative expressions are evaluated relative to the context node, and absolute expressions are evaluated with the context node-set to the root of the XPath tree.

The Full Form of Path Expressions

Here is an example of an absolute full-form path expression:

```
/child::play/child::scene[position()=3]/child::speech[attribute::speaker="King"]
```

Stepping through the expression from left to right, we first select all `play` elements that are children of the root (of which there can be only one, according to XML rules). We next select the third `scene` element that is a child of `play`, and finally all the `speech` elements that are children of the third `scene` and have

a `speaker` attribute whose value is "King". Sometimes, reading from right to left is clearer: this location path selects all the speeches by King in the third scene of the play.

Location Steps and Their Components

A path expression consists of one or more steps separated by a "`/`". Our example has three location steps. A location step consists of three parts (two required and one optional):

- *axis*: In what direction are we going?
- *node test*: What nodes are we selecting while going in that direction?
- *additional predicate (optional)*: further conditions on nodes to be selected.

The axis name is separated from the node test by the "::" separator. The additional predicate, if any, appears in square brackets.

Node Tests

A node test is either a name test or a node-type test. The simplest name test is a literal, the name of an element or attribute node. For instance, `child::speech` selects all the children of the context node that are speech elements, and `attribute::speaker` selects all the attributes of the context node whose name is `speaker` (of which there can be at most one, according to XML rules). The expression

```
child::speech/attribute::speaker
```

selects all `speaker` attributes of all `speech` elements that are children of the context node. If the context node is a `scene`, the expression will select all the speakers within that scene, with repetitions. In the next chapter, we will explain several ways to select a list without repetitions and compare them for efficiency.

*The * Node Test*

The * node test matches any name, but only within a certain type, depending on the axis. Each axis has a principal node type, defined as follows: for attribute and

namespace axes, the principal node types are attribute and namespace, respectively; for all other axes, it is the element node type. So,

```
/descendant::scene[position()=3]/preceding-sibling::*
```

selects all the element nodes that are siblings of the third scene and precede it in the document order, whereas

```
/descendant::scene[position()=3]/attribute::*
```

selects all the attributes of the third scene.

Remember that a node-set can contain nodes that have no structural relation within a tree. The following two expressions select the same node-set only if all the siblings of the third `annotation` element are also `annotation` elements:

```
/descendant::annotation[position()=3]/following-sibling::*
/descendant::annotation[position()>=3]
```

Even if they select the same node-set, the meanings of the two expressions are different: the first selects in terms of tree structure (following siblings of the third annotation, whatever their tag name), and the second selects within a linear node-set of annotations (all annotations from third to last, in document order).

*The * Test and the Namespace*

The * test can be used in combination with a namespace. Suppose that the root element of a document carries this namespace declaration:

```
xmlns:art="http://www.n-topus.com/ns/arts"
```

The expression `/descendant::art:*` will select all elements in the document that belong to that namespace. It is the namespace that matters, not the specific prefix. If elsewhere in the document the same namespace is mapped to a different prefix, the elements with that prefix will still match the expression. Similarly,

```
/descendant::*/attribute::xlink:*
```

will select all `xlink` attributes in the document.

`node()`, `text()`, `comment()`, ***and*** `processing-instruction()`

The node test `node()` is true for any node of any type. Whereas `child::*` selects only children that are elements, `child::node()` will also select text nodes,

comment nodes, and processing-instruction nodes (but not attribute nodes or namespace nodes).

To select text, comment, or processing-instruction nodes specifically, you use `text()`, `comment()`, and `processing-instruction()`, respectively. The expression

```
/descendant::comment()
```

will select all the comment nodes in the document.

Additional Predicates

The optional third component of a location step within a path expression is an additional predicate. It can be any Boolean expression. If the expression in brackets evaluates to a non-Boolean object, the object is converted to a Boolean as we discussed. For instance, to select only elements that have an `xlink:type` attribute, you say

```
/descendant::*[attribute::xlink:type]
```

The path expression in brackets evaluates to a node-set that is converted to False if the node-set is empty and to True otherwise. Similarly, the expression

```
/descendant::*[not(node())]
```

selects all element nodes that have no children. These are empty-element nodes, possibly with attributes but without text.

Equality and Inequality Operators with Node-Sets

Additional predicates within path expressions frequently compare node-sets. When the `=` operator is applied to two node-sets, its meaning in XPath is somewhat unusual: the entire expression evaluates to True if the intersection of the two node-sets is not empty. In other words, if there exists a node N1 in node-set NS1 and node N2 in node-set NS2 such that `string(N1)=string(N2)`, then `"NS1=NS2"` evaluates to True. One consequence of this is that, if one of the two node-sets is empty, the equality expression always evaluates to False, even if both node-sets are empty. There is no operator that would check to see whether all nodes in NS1 and NS2 are equal. In *XPath 2.0*, both operations will be supported.

A common example of comparing node-sets for equality is in outputting a set of unique references from data that contains repetitions. Recall the data of Listing 5-1:

```
<bible-linkbase-src>
 <ref>
   <from>nt.xml Luke 18 27</from>
   <to>ot.xml Genesis 18 14</to>
 </ref>
 <ref>
<!-- more ref elements -->
</bible-linkbase-src>
```

The following template, `within"bibleLinks/noRepRefs.xsl"`, will output all verse references in bibleLinks/biblelinks.xml without repetitions:

```
<xsl:template match="ref/to | ref/from">
  <xsl:if test="not(. = preceding::from | preceding::to)" >
    <p><xsl:value-of select="." /></p>
  </xsl:if>
</xsl:template>
```

You can test it directly as

```
http://localhost:8080/xmlp/xx.jsp?
    xmlUri=bibleLinks/biblelinks.xml&
    xslUri=bibleLinks/noRepRefs.xsl&
    method=html
```

This code is used in the XLink application of this chapter. In the next chapter, we will show how to perform the same operation much more efficiently.

Union Expressions

The "|" operator forms union expressions. The expression

```
/descendant-or-self::*|/descendant-or-self::*/attribute::*
```

will select all element and attribute nodes in the document. In most stylesheets, it will be written in its abbreviated form:

```
//*|@*
```

As you can see, the keystroke savings (and, in many cases, the improvement in readability) can be substantial. The next section introduces the abbreviated form.

Abbreviated Form of Path Expressions

Predicates, axis specifiers, steps, and entire path expressions have abbreviated forms.

Abbreviated Predicates

Predicates of the form `[position()=3]` can be abbreviated to `[3]`.

Abbreviated Axis Specifiers

Abbreviations are provided for the two most common axes: `child` and `attribute`. If the axis specifier is abbreviated completely out of existence, the default `child` axis is assumed. The `attribute::` specifier is abbreviated to `@`. Here is an earlier example followed by its abbreviated form:

```
/child::play/child::scene[position()=3]/child::speech[attribute::speaker="King"]
/play/scene[3]/speech[@speaker="King"]
```

Abbreviated Steps

Two directory path conventions—"." for the current directory and ".." for the parent directory—are adopted to represent `self::node()` and `parent::node()`, respectively. For instance, this will select all descendants of the context node that have both attributes `a1` and `a2`:

```
./descendant-or-self::node()/*[@a1 and @a]
```

The next example returns True if there is no parent node. This is the way to test whether a given node is the root. (There is no `root()` predicate in XPath.)

```
not(..)
```

To find the position of the parent node within the list of its siblings, you can say

```
count(../preceding-sibling::node()) + 1
```

Note that `count()` is a node-set function: it takes a node-set and returns the number of nodes in it.

Abbreviated Paths

The abbreviation `//eltname` stands for `/descendant-or-self::node()/eltname`. It can be used in an absolute or relative expression. Here are some examples:

- `//eltname`: all element nodes whose name is `eltname`
- `//*[@xlink:type]`: all elements that have an `xlink:type` attribute
- `.//*[@a1 and @a2]`: all element nodes within the context node subtree (including the context node itself) that have both `a1` and `a2` attributes

The `//` abbreviation is a quick but inefficient way of specifying a node-set: use it during the design and initial development stage but look for ways to optimize it before you are done. More specific expressions are always preferable: if all `speech` elements are found at `/play/scene/speech`, then this expression, or even `/play/*/speech`, will be considerably more efficient than `//speech`. Other optimizations are discussed later in the chapter.

Summary of Abbreviations and Wildcards

The following list summarizes all the abbreviations introduced in this section.

- If no axis is shown, the `child::` axis is assumed.
- `attribute::` abbreviates to @.
- `[position()=5]` abbreviates to `[5]`.
- * stands for all element children of context node.
- `@*` stands for all attributes of context node.

- `//d` stands for `/descendant-or-self::node()/child::d/` (that is, all `d` descendants of root).
- `.` stands for the context node.
- `.//d` stands for all `d` descendants of context node.
- `..` stands for the parent of context node.

Node-Set Functions

Node-set functions fall into three groups: those that evaluate to a number, those that evaluate to a string (name functions), and the `id()` function that evaluates to a node-set. We present them in that order. The number-valued functions are

- `last()`, no arguments: the number of nodes in the node-set
- `count(node-set)`: the number of nodes in the argument node-set
- `position()`, no arguments: the position of the context node

All string-valued functions take an optional node-set argument. (The familiar "?" notation is used to indicate that the argument is optional.) If the argument is omitted, it defaults to a node-set with the context node as its only member. In either case, the function is applied to the first node of the node-set.

- local-name(*node-set*?): the local name of the first node
- namespace-uri(*node-set*?): the namespace URI of the first node
- name(*node-set*?): concatenated namespace URI (if any) and the local name

Finally, the `id()` function takes a string argument that can be a single token or a list of tokens. For each token in its argument, the function looks for an element node that has an attribute of type ID, whose value is equal to the token. Remember that an attribute of type ID does not have to be named id, but it must be declared as type ID in the DTD. In other words, you must have a DTD if you use the `id()` function, but the good news is that you don't have to validate. Most parsers (and all those that you are likely to use) will process attribute declarations even if they are not asked to validate, so your DTD can consist just of those declarations. (See Listing 5-5 for an example.)

Listing 5-5. People with IDs

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE pdata [
<!ELEMENT person ANY>
<!ATTLIST person id ID #REQUIRED>
]>
<pdata>
  <person id="A2">
    <name>
      <title>Mr.</title><last>Bargle</last><first>Bertrand</first>
    </name>
    <bdate>
      <year>1958</year> <month>12</month><day>30</day>
    </bdate>
  </person>
  <!-- many more persons -->
</pdata>
```

With a DTD like that, you can use the `id()` function to locate a specific person. This is frequently the most efficient way to locate a node because most XSLT processors will compile IDs into a hashtable. With a single ID as argument, `id("A3")` is equivalent to `//person[@id="A3"]`. With more than one token, you get the union of the results: `id("A3 A4")` is equivalent to `//person[@id="A3" or @id="A4"]`. You may want to run xptrans.jsp on a couple of examples to see how the `id()` function works. For instance, enter "/helloXSL/pdata2DTD.xml" as the URI and `id("CM123")` as the XPath (keeping default values for the other two parameters), and you will see the Cookie Monster record.

> **NOTE** *It is common to estimate the efficiency of a procedure as a function of the size of its input. If we were to look for a specific person node by checking each such node, the time it would take would be in proportion to the length of the list of persons—in proportion to one-half the size of the list, on the average. Getting the node from a hashtable takes a constant time. If the list is long, this makes a big difference.*

XSLT Processing Model

An XSLT stylesheet operates on the XPath tree of the source document. Its main processing component is a template rule, similar to a function or subroutine in other languages. In terms of document structure, an XSLT stylesheet proceeds like this:

- the start tag of the root element that declares namespaces and specifies the version
- top-level elements (that is, children of the root) other than template rules
- one or more template rules that contain literal result elements and instruction elements
- the closing tag

The Template Rule, an Example, and a Summary

A template rule has two parts: a pattern that is matched against nodes in the source tree, and a template body that gets instantiated to form part of the result tree. In XML terms, the `xsl:template` element has the mixed-content model: the template body consists of elements and text. In XPath terms, an `xsl:template` element node has element node children and text node children. All text nodes and non-XSLT element nodes are passed to the result tree unchanged. XSLT children of `xsl:template` are interpreted as instructions to be carried out by the XSLT processor. The official terminology is *literal result elements* for non-XSLT material and *instruction elements* for XSLT children of `xsl:template`. Listing 5-6 provides an example to illustrate these concepts and terminology.

Listing 5-6. Stylesheet Components

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:p="http://www.n-topus.com/ns/samples/X"
  version="1.0"><!-- end of start tag of root element -->
<xsl:output method="html"/><!-- top-level element -->
<xsl:strip-space elements="*"/><!-- top-level element -->
<xsl:template match="/" ><!-- start tag of template rule -->
                         <!-- matches the root of source -->
<html><!-- this and the next 3 lines are literal result elements -->
```

```
  <head><title>Select 1</title></head>
  <body><h3>Select 1</h3>
  <div>
      <xsl:value-of select="/p:exchange/p:q[1]" />
      <!-- XSLT instruction:
      output the first question -->
  </div></body></html><!-- end tags of literal result elements -->
</xsl:template><!-- end tag of template rule -->
</xsl:stylesheet><!-- end tag of root element -->
```

When applied to the document of Listing 5-7, this stylesheet produces the HTML page of Listing 5-8; in other words, Listing 5-8 comes from

```
http://localhost:8080/xmlp/xx.jsp?
    xmlUri=moreXSL/Xchange.xml&
    xslUri=moreXSL/firstQuestion.xsl&method=html
```

Note the treatment of namespaces: the same namespace URI is the default (no prefix) namespace in the XML file but mapped to the `p:` prefix in the stylesheet. However, the unprefixed `q` element in the data is matched by the `p:q` element in the stylesheet because it's the namespace URI, not the prefix, that is used in matching.

Listing 5-7. The Source Document

```
<?xml version="1.0"?>
<exchange tone="informal" xmlns="http://www.n-topus.com/ns/samples/X">
  <q>What's up?</q><a>Nothing much. </a>
</exchange>
```

Listing 5-8. The Output

```
<html xmlns:p="http://www.n-topus.com/ns/samples/X">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Select 1</title>
   </head>
   <body>
      <h3>Select 1</h3>
      <div>What's up?</div>
   </body>
</html>
```

Top-Level Elements Summarized

In addition to `xsl:template`, the following more commonly used top-level elements are discussed and used in this book:

- `import` and `include`: includes material from another stylesheet
- `param` and `variable`: associates a name with a value (`param` is settable from outside the stylesheet, such as from the command line)
- `output`: sets output method to either XML, HTML, or text
- `strip-space` and `preserve-space`: controls transfer of whitespace-only nodes (for example, whitespace used for formatting) from input document to result tree
- `key`: declares a key table (usually implemented as a hashtable) to use in search

See Kay's book for the remaining top-level elements:

- `attribute-set`: defines a named set of attributes
- `namespace-alias`: declares an alias within the stylesheet for a namespace in result tree
- `number`: inserts a formatted number into result tree; number siblings
- `decimal-format`: works with format-number

Instruction Elements Summarized

The following types of instruction elements are discussed and used in this book. (Elements in parentheses indicate commonly used children of instruction elements.)

- *Push and pull control of evaluation*: `apply-templates`, `for-each`
- *Other control of evaluation*: `call-template` (`with-param`), `apply-imports`
- *Output by construction*: `element`, `attribute`, `comment`, `processing-instruction`

- *Output marked-up content by copying*: `copy` (shallow copy), `copy-of` (deep copy)
- *Output a value*: `value-of`
- *Conditionals*: `if`, `choose` (`when`)
- *Output a message and optionally terminate processing*: `message`

See Kay's book for the remaining instruction element, `fallback`, that is used for graceful degradation and future proofing.

Order of Evaluation: Pull and Push

The first template to be instantiated is always a template that matches the root element. If no such template is provided, then the default root-matching template is used, as explained in the section on default templates. For now, let's assume that the root is matched (as in Listing 5-6), and the body of the template gets instantiated. In the process, additional template bodies can get instantiated and included in the result tree as the result of evaluating two instructions: `xsl:for-each` or `xsl:apply-templates`. (There is also `xsl:apply-imports`, but it is rarely used.)

`xsl:for-each` and `xsl:apply-templates` are XSLT analogs of the loop and the subroutine call, except that both are data-driven: they establish a list of nodes to process, and, for each node on the list, they provide a template body to instantiate. An `xsl:for-each` element provides that template body directly as its own element content, within the same template rule. In Listing 5-9 (which is identical to Listing 5-3 but with additional comments), the contents of the `xsl:for-each` element are instantiated for each person's record in the "People with IDs" document of Listing 5-5.

`xsl:apply-templates`, shown in Listing 5-10, sends each node on the current node list to look for a template that will match it and provide a template body. Because `xsl:for-each` pulls the nested template bodies into its own template, its use is called *pull processing*, and using `xsl:apply-templates` is called *push processing* because the current node list is pushed out of the current template into other templates.

The next two listings show two stylesheets that do exactly the same thing, with one using pull and the other push. Both are intended for the "People with IDs" document of Listing 5-5. The nested template body is highlighted in both.

Listing 5-9. Names in a Table Using Pull

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/" ><!-- start tag of template rule -->
                       <!-- matches the root of source -->
  <html><head><title>People's names</title></head>
                       <!-- literal result elements -->
    <body><table border="1"><!-- literal result elements -->
     <xsl:for-each select="people/person/name">
<!-- start of template body to be instantiated for each name element -->
      <tr>
         <td><xsl:value-of select="title"/></td>
         <td><xsl:value-of select="first"/></td>
         <td><xsl:value-of select="last"/></td>
      </tr>
 <!-- end of template body to be instantiated for each name element -->
    </xsl:for-each>
   </table></body>  </html>
</xsl:template>
</xsl:stylesheet>
```

Listing 5-10. Names in a Table Using Push

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/" >
  <html><head><title>People's names</title></head>
    <body><table border="1">
      <xsl:apply-templates select="pdata/person/name" />
    </table></body>
  </html>
</xsl:template>
<xsl:template match="name" >
<!-- start of template body to be instantiated for each name element -->
  <tr>
   <td><xsl:value-of select="title" /></td>
   <td><xsl:value-of select="first" /></td>
   <td><xsl:value-of select="last" /></td>
  </tr>
<!-- end of template body to be instantiated for each name element -->
</xsl:template>
</xsl:stylesheet>
```

Both stylesheets output an HTML page that puts the names in a table, as shown here:

```
<html><head> . . . </head>
   <body>
      <table border="1">
         <tr>
            <td>Mr.</td>
            <td>Bertrand</td>
            <td>Bargle</td>
         </tr>
...
</table></body></html>
```

Push and Pull Without a select *Attribute*

If there is no `select` attribute, both `xsl:for-each` and `xsl:apply-templates` default to using the children of the current node as the current node list. This usage is very common in push stylesheets that recursively descend into the document tree, providing a template for each element type in the document. Here is Listing 5-10 rewritten in this style.

Listing 5-12. Push Without `select`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/" ><!-- match root, output what's needed at this stage -->
 <html><head><title>Helpers</title></head>
   <body>
    <xsl:apply-templates /><!-- apply templates to children -->
   </body>
 </html>
</xsl:template>
<xsl:template match="pdata" >
 <table border="1">
 <xsl:apply-templates />
 </table>
</xsl:template>
<xsl:template match="person" ><!-- output a row for each person -->
 <tr>
 <xsl:apply-templates />
 </tr>
</xsl:template>
<xsl:template match="name" >
```

```
 <xsl:apply-templates />
</xsl:template>
<xsl:template match="title | last | first" >
     <!-- output each of these in a td -->
 <td><xsl:apply-templates /></td>
</xsl:template>
<xsl:template match="bdate" >
</xsl:template>
</xsl:stylesheet>
```

The last two templates (in boldface) require comment. The first of them contains a call on `xsl:apply-templates` for which no explicit templates are provided. The rest of the processing is done by default templates. First, a default template calls `apply-template` on the children of `title`, `last`, and `first`, because they are element nodes. They all have a single child that is a text node. The default template for text nodes outputs their content into the result tree.

The last (empty) template is needed to override default processing of the `bdate` element nodes. Without it, the default templates would recursively descend into those nodes, reach the text nodes, and output their contents, unformatted, into the result tree.

The output of this stylesheet is different from the preceding one because we do not arrange the order of cells within a row ourselves—so they get arranged in document order, which is title, last, first. To change that, we would rewrite the template for `name`, pulling its contents in by means of `xsl:value-of`:

```
<xsl:template match="name" >
 <!-- replace xsl:apply-templates with: -->
   <td><xsl:value-of select="title" /></td>
   <td><xsl:value-of select="first" /></td>
   <td><xsl:value-of select="last" /></td>
</xsl:template>
```

Sorting

Both `xsl:for-each` and `xsl:apply-templates` can have an `xsl:sort` child element that specifies the order in which the current node list is processed; if no such element is present, the list is processed in document order. Suppose we want to list our people alphabetically by last name. Each person has a row, so we give an

`xsl:sort` child to that instance of `xsl:apply-templates` that gets matched by person elements, within the template that matches `pdata`:

```
<xsl:template match="pdata" >
 <table border="1">
 <xsl:apply-templates>
   <xsl:sort select="last"/>
 </xsl:apply-templates>
 </table>
</xsl:template>
```

We can achieve the same result in our pull stylesheet by rearranging its `xsl:for-each` element like this:

```
<xsl:for-each select="pdata/person">
 <xsl:sort select="name/last"/>
 <tr>
    <td><xsl:value-of select="name/title"/></td>
    <td><xsl:value-of select="name/first"/></td>
    <td><xsl:value-of select="name/last"/></td>
 </tr>
</xsl:for-each>
```

You can sort alphabetically or numerically, in ascending or descending order. To sort our persons in the descending order of their year of birth, you would say

```
<xsl:for-each select="pdata/person">
  <xsl:sort select="bdate/year" data-type = "number" order="descending" />
</xsl:for-each>
```

Using Push with a `mode` *Attribute*

It frequently happens that you want to process the same set of nodes more than once, perhaps to present two different views of the data or to compile an index before displaying full text. In our XLink application, an input `ref` element such as

```
<ref>
   <from>nt.xml Luke 18 27</from>
   <to>ot.xml Genesis 18 14</to>
 </ref>
```

has to be processed twice, first to construct locator elements, and then to construct arcs. We need two templates that match exactly the same data but have different template bodies. To distinguish them, both `xsl:apply-templates` and `xsl:template` have another attribute called `mode`. If `xsl:apply-templates` specifies a `mode`, only templates that have the same value of `mode` will match. Our ref2link.xsl is structured as follows:

```
<xsl:template match="/">
  <elt xlink:type="extended">
    <xsl:apply-templates mode="locators"/>
    <xsl:apply-templates mode="links"/>
  </elt>
</xsl:template>

<xsl:template match="ref" mode="arcs">
<!-- create arcs elements -->
<xsl:template match="ref/to | ref/from" mode="locators">
<!-- create locator elements -->
```

The `mode` attribute can be used in many interesting ways, some of them quite intricate (not to say tricky), but the main use as illustrated here is completely straightforward.

Push and Pull Contrasted

If push and pull are so similar, why do we need both? Push and pull work together in XSLT in much the same way that event-driven and procedural code work together in more conventional languages. A push template is like an event handler, triggered by the occurrence of a matching block of XML. The explicit algorithmic control offered by `xsl:if`, `xsl:choose`, `xsl:for-each`, and `xsl:call-template` closely parallels the corresponding structures in C++ or Java. For some problems, very clean solutions are available that will take the form of pure push; other problems can be solved better with pure pull. Usually, we end up with some kind of a mix, as in Listing 5-10, in which most processing is pushed out to other templates but in the end we pull in values using `xsl:value-of`.

In pull code, especially with the use of `xsl:for-each` rather than template matching, the XSLT structure mirrors the XML structure, so that someone reading the XSLT can tell what the XML was to look like. In contrast, the purest push templates will simply say what should happen at the bottom level; the XML structural information is taken from the XML itself at runtime. Each of these has advantages in terms of clarity and maintainability, and the choice frequently boils down to individual taste.

Default Templates and Conflict Resolution

When a node is sent out to look for a template that will match and process it, three outcomes are possible:

- Exactly one template matches the node.
- No templates match the node.
- More than one template matches the node.

The first case is easy. In the second case, a default template is applied. In the third case, the processor uses the standard conflict-resolution policy to select the best match.

Default Templates

Each of the seven node-types has a default template, as shown in Table 5-2. The attribute default rarely gets used because attributes are not children of their element, and it takes a special effort to get to them. Defaults for element and text nodes are used routinely to output text values.

Table 5-2. Node Types and Default Templates

NODE TYPE	DEFAULT TEMPLATE ACTION
root	Apply available templates to children, in document order
element	Apply available templates to children, in document order
text	Copy the text content
attribute	Copy the attribute value as text
comment	Do nothing
PI	Do nothing
namespace	Do nothing

Precedence Rules

More than one template rule may be applicable to a node, in which case precedence rules are used to determine which template rule to apply. Precedence is mostly based on two considerations:

- Templates in imported stylesheets have lower precedence than do templates in stylesheets that import them.
- More-generic templates have lower precedence than do more-specific templates: a template with `match="*"` that matches elements of any name has lower precedence than does a template with `match="title"` that matches only title elements.

The `priority` attribute of `xsl:template` can override other considerations. However, if your program crucially relies on that attribute for its control structure, then either your program can be improved or you are using XSLT for a task that it was not designed for.

`xsl:include`, `xsl:import`, *and* `xsl:apply-imports`

XSLT has two top-level elements that bring in material from another stylesheet: `xsl:include` and `xsl:import`. Both have an `href` attribute to specify the URI of the external stylesheet. `xsl:include` does straight inclusion, as if the included material were physically present in the including document. For conflict resolution, included templates have the same precedence as home templates. xsl:import brings in material that can be overridden: if an imported template matches the same element as a home template, then the home template always wins, no matter how generic it is. The only way to apply an imported template rule that is shadowed by a home template that matches the same nodes is by means of `xsl:apply-imports`.

Generic and Specific

Patterns have several gradations of specificity. The two most important distinctions are as follows.

- Patterns that use node classes (such as `node()`, `text()`, *) are less specific and have lower precedence than patterns that use node names.
- Patterns with additional predicates are more specific (other things being equal) than patterns without additional predicates.

Parameters, Variables, and Result Tree Fragments

XSLT parameters and variables have a good deal in common. Both `xsl:param` and `xsl:variable` are used to associate a name with a value. The ways values are specified are exactly the same.

Specifying a Value

The value can be specified in two ways: as a template body supplied as the content of the `param/variable` element or as an XPath expression supplied as the value of the `select` attribute. If there is a `select` attribute, the element must be empty:

```
<xsl:param name="example" select="//person/name" /><!-- $example is a node-set -->
<xsl:variable name="str" select="'a literal string'"/><!-- $str is a string -->
<xsl:param name="num" select="275*3-23"/><!-- $num is a number -->
<xsl:param name="anotherExample">
  <xsl:copy-of select="//person/name" />
</xsl:param>
```

If the value is specified using `select`, the data type of the value will be one of the four XPath data types, depending on the expression. Note that, if you want to supply the value as a literal string, you need two levels of quotation marks:

```
<xsl:param name="country" select="'Benin'"/>
```

If you omit the inner quotes, the processor will look for the child of the current element whose name is “Benin”. The outer quote marks are for the XML parser; the inner quote marks are for the XSLT/XPath processor.

Result Tree Fragment

If the value of a variable or parameter is specified by a `select` attribute, the data type of that value is one of the four XPath types: either a scalar type (Boolean, string, or number) or a node-set extracted from the input tree. If the value is specified as a template body, its data type is the XSLT-specific result tree fragment. It is a temporary tree structure that is quite similar to the regular XPath tree except the root can have more than one element child. In most cases, the tree is

automatically converted to a simple value (string, number, or Boolean), and you don't have to think about it. However, a problem arises if you want to process it as a node-set because *XSLT 1.0* provides no function for converting a result tree fragment to a node-set. This problem will go away in *XSLT 2.0* because there will be no additional result tree fragment data type. In the meantime, every major XSLT processor has provided an extension function for converting result tree fragments to node-sets. We show how they work in the next section.

Position and Usage

Parameters and variables can be either top-level or local to a template. Within a template, variables can appear anywhere, but parameters must be listed in the very beginning, before anything else.

Parameters Are for Passing Parameters

Variables are to hold a value for future reference, and parameters are for receiving parameter values either to the entire stylesheet (top-level parameters) or to a template (template-level parameters). In either case, the values of parameters are defaults that can be overridden by values supplied from the outside.

The way parameters are passed to the stylesheet is implementation specific. From a command line, name-value pairs can usually be entered after the other arguments. This is the syntax used by Michael Kay's Saxon processor:

```
saxon countries.xml econsummary.xsl > beninecon.html country=benin
```

In a Web application, stylesheet parameters can be passed in the usual way from an HTML form using GET or POST. You have seen examples in the xx.jsp application (Listing 1-10 of Chapter 1); the webapp simply looks at its parameters and uses the implementation-specific stylesheet parameter-setting mechanism to pass them along. Within JAXP, we say

```
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer(new StreamSource(xslUri));
transformer.setParameter(nameString,valString);
```

The way parameters are passed from one template to another within a stylesheet is precisely defined in the specification. This is the subject of the next section.

Named Templates and Recursion

To pass parameters (and control of execution) from one template to another, the receiving template must be a named template: it must have a `name` attribute whose value is the name of the template. The parameters of a named template are `xsl:param` children of a named template. They must be listed in the very beginning of the template's body, and they can be given default values in select attributes:

```
<xsl:template name="list-helpers">
  <xsl:param name="theList" select="''"/>
  <!-- the rest of template body -->
</xsl:template>
```

To pass control to a named template, you use `xsl:call-template`. (See the example in the next section.) You can pass along additional parameters using `xsl:with-param` children elements of `xsl:call-template`. Their values will override the defaults (if any) that are specified in `xsl:param` children of the named template.

Compared to the `xsl:apply-templates` instruction, `xsl:call-template` feels much more like a "normal" procedure call. Instead of passing control in a diffuse way to "a highest-precedence template that matches this node," you can pass control and parameters to a specific template, identified by name. A template must have either a `match` attribute or a `name` attribute, and it may have both.

A Simple Example: Tables of Helpfulness

We illustrate common uses of `call-template` with an example based on the people data of Listing 5-5. We have simplified the data by removing titles and birthdays. Instead, we have given each person another attribute, a list of the people whom that person finds helpful in his or her work. (The attribute is used in assigning people to cubicles in such a way as to maximize total mutual help.)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<pdata>
  <person id="A2" helpers="A3 A4 A5">
    <name><title>Mr.</title> <last>Bargle</last><first>Bertrand</first></name>
  </person>
  <!-- many more persons -->
</pdata>
```

Our stylesheet prints a table of helpers for each person.

Figure 5-5. Tables of helpers

In the first version, `xsl:call-template` is used simply as a subroutine call, to break up what would otherwise be a large chunk of code. The code is concise enough to see all at once, and it is shown in Listing 5-12.

Listing 5-12. Tables of Helpers

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/" ><!-- start top-level template matching the root -->
 <html><head><title>Tables of helpfulness</title></head><body>
 <xsl:for-each select="/pdata/person">
   <xsl:variable name="myHelpers" select="@helpers"/>
   <xsl:variable name="myName" select='concat(name/first," ",name/last)'/>
   <p>Here are the people helpful to <xsl:value-of select="$myName"/>:
<!-- call named template to output a person's list of helpers as a table -->
     <xsl:call-template name="list-helpers">
       <xsl:with-param name="theList" select="$myHelpers" />
     </xsl:call-template>
    </p>
<!-- end of list of helpers output -->
 </xsl:for-each>
</body></html>
</xsl:template>

<xsl:template name="list-helpers">
  <xsl:param name="theList" select="''"/><!-- parameter declared -->
  <table border="1">
```

```
    <xsl:for-each select="//person"><!-- select all person elements -->
      <xsl:if test="contains($theList,@id)">
        <tr>
          <td><xsl:value-of select="name/last"/></td>
          <td><xsl:value-of select='name/first'/></td>
        </tr>
      </xsl:if>
    </xsl:for-each>
  </table>
</xsl:template><!-- end of named template -->
</xsl:stylesheet>
```

Two Efficiency Improvements

The `list-helpers` template does a search through the list of persons (obtained by `//person`) to find the person with a given ID. We can improve it in two ways: replace `//person` with `/pdata/person` and replace `xsl:if` with a predicate:

```
    <xsl:for-each select="/pdata/person[contains($theList,@id)]">
        <tr><!-- exactly as before --></tr>
    </xsl:for-each>
```

Replacing `//person` with a more specific path is almost always an improvement, and sometimes a very substantial one. In the next chapter, we will show timings to support this claim, and we will show how to calculate those timings. Replacing `xsl:if` with an XPath predicate is likely to be an improvement because the predicate query is likely to be optimized. The query with `xsl:if`, if taken literally, asks the processor to build the list of all persons and then filter that list. With the XPath predicate, the processor probably does not build the intermediate list.

Even with this improvement, each call on `contains()` takes time proportional to the number of persons, and, because we do it for each person, each call on `list-helpers` takes time proportional to the square of the number of persons. Put differently, if the number of persons is doubled, the time is quadrupled. A significant improvement would be to do `list-persons` in linear time. This can make a difference between doable and undoable for a list of tens of thousands of people. One way to achieve linear-time performance is to replace `contains()` that takes linear time with `id()` that takes constant time. (For `id()` to work, we will need a minimal DTD as discussed in the earlier section on node-set functions.) This change is implemented in the next section.

Recursive Processing of a List of Tokens

In Listing 5-11, we process a list of nodes (the current node list) and retrieve the ID for each node. In the next version, we will process a list of IDs directly, as a string that is a space-separated list of tokens. This is a very common control structure in XSLT: repeat the same action for each token in a string. A token is assumed to be of the XML NMTOKEN data type or its equivalent in XML Schema; in particular, a token does not contain whitespace. We also assume that the list of tokens is normalized, in the *XML 1.0* sense of the term, that is, that the tokens are separated by a single space character and there is no whitespace before the first token or after the last one.

To make the processing uniform, there is a simple gimmick that we first used in our 1999 Wrox book, *Professional Java XML Programming*: terminate the string with a space character so that every token in the string, including the last one, is followed by a space. In this stylesheet, we do it in the definition of the `helpers` variable:

```
<xsl:variable name="myHelpers" select="concat(@helpers,' ')"/>
```

As before, we pass the variable as a parameter to `list-helpers`. What follows is different: list-helpers outputs only the table tags but otherwise serves as a wrapper for `make-rows` that does all the work:

```
<xsl:template name="list-helpers">
  <xsl:param name="theList" select="''"/>
  <table border="1">
     <xsl:call-template name="make-rows">
        <xsl:with-param name="theList" select="$theList" />
     </xsl:call-template>
  </table>
</xsl:template>
```

The logic of `make-rows` is simple: for each token in `theList`, find the corresponding node and output it as a row. In many programming languages, this task would be done by a loop that may be described in pseudo-code this way:

```
current = firstToken(); do makeRow(); current=nextToken(); until noMoreTokens();
```

The problem with writing this out in XSLT is that XSLT has no assignment operator. You can create a variable with a certain value but there is no way to change that value. We will discuss the motivations for this design decision in the next chapter; our task now is to implement `make-rows`. In languages without

assignment, this is done using recursion rather than a loop. The logic is as follows:

```
if theList is not empty
  call make-row with the first token of theList
  call make-rows recursively with tail-end of theList, without the first token
otherwise
   do nothing
```

In XSLT, this comes out as shown in Listing 5-13.

Listing 5-13. List of Tokens Recursive Pattern

```
<xsl:template name="make-rows">
  <xsl:param name="theList" select="''"/>
  <xsl:if test="string-length($theList)!=0"><!-- if theList is non-empty -->
    <xsl:call-template name="make-row">
      <xsl:with-param name="friend" select="substring-before($theList,' ')" />
    </xsl:call-template>
    <xsl:call-template name="make-rows">
      <xsl:with-param name="theList" select="substring-after($theList,' ')" />
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

This kind of linear recursion down the space-terminated list of space-separated tokens is a very common programming pattern in XSLT.

All that remains is to implement `make-row`, given an ID. We can do it in constant time, using the `id()` function:

```
<xsl:template name="make-row">
  <xsl:param name="friend" select="''"/>
  <xsl:for-each select="id($friend)">
      <tr>
        <td><xsl:value-of select="name/last"/></td>
        <td><xsl:value-of select='name/first'/></td>
      </tr>
  </xsl:for-each>
</xsl:template>
```

The `make-rows` template—and therefore the entire stylesheet—now takes time that is linear in the size of output, a major improvement over the preceding version.

Variables, Result Tree Fragments, and Node-Sets

You may be wondering why we have not implemented `make-row` using a variable:

```
<xsl:template name="make-row">
  <xsl:param name="friend" select="''"/>
  <xsl:variable name="current" select="id($friend)">
  <tr>
     <td><xsl:value-of select="$current/name/last"/></td><!-- will not work -->
     <td><xsl:value-of select='$current/name/first'/></td><!-- will not work -->
   </tr>
  </xsl:for-each>
</xsl:template>
```

This won't work because the value stored in the `current` variable is not a node-set, but rather a result tree fragment, a data type that is specific to *XSLT 1.0*. Moreover, *XSLT 1.0* has no standard function to convert result tree fragments to node-sets. However, a need for such a function arises so often that every major processor has implemented it. (In *XSLT 2.0*, the result tree data type and the entire problem will go away.) To use a conversion function, you have to include some processor-specific code, but all processors follow the same conventions: you have to declare a processor-specific namespace and invoke the function using the namespace prefix. For instance, to use Xalan, you would add the following namespace declaration to the root element:

```
xmlns:xalan="http://xml.apache.org/xalan"
```

To fix this broken code, you would write

```
<td><xsl:value-of select="xalan:nodeset($current)/name/last"/></td>
```

To switch to the Microsoft processor, you would need to declare another namespace and replace the prefix in the code. Unfortunately, you would also need to insert a hyphen into `nodeset`:

```
xmlns:msxml="urn:schemas-microsoft-com:xslt"
<td><xsl:value-of select="msxml:node-set($current)/name/last"/></td>
```

It makes sense to include namespace declarations for all of the processors that you are likely to use and just change the prefixes in your code when you switch processors. This minor hassle will disappear with the next release of XPath.

The Code of the XLink Application

We now have all the tools to go through the code of the XLink application. As explained in the beginning of the chapter, the application consists of two parts. Part 1, ref2xlink.xsl, creates a linkbase from an XML "link source" file that itself can be easily created from user input via an HTML form. The linkbase contains extended link structures that describe many-to-many links in XML data. Part 2, open-ended, implements queries. In this section, we show only one query, neighbors.xsl, that finds the neighbors of a Bible verse in the XLink graph.

We present ref2link.xsl first. The input and output formats are shown in Listings 5-1 and 5-2 in the beginning of the chapter. (If you need a reminder of what an XPointer looks like, review Listing 5-2.) Our task is to transform each `ref` element of Listing 5-1 into an `arc` and two `locator` elements of Listing 5-2, except we want to output each locator only once even if it is mentioned in several `ref` elements on input.

Link Source to Linkbase Transformer

Before looking at the code, let's review some of the tasks it has to perform.

- Go over the same `ref` element twice, first to construct locators, then to construct arcs.
- In constructing locator labels, convert a string like "nt.xml Luke 18 27" into "Luke_18_27". The same labels are used in `arc` elements.
- In constructing locator hrefs, convert a string like "nt.xml Luke 18 27" into a complete URL+XPointer expression.
- For either labels or hrefs, one of the tasks is to break the original string into pieces using the space character as the separator.

Control Structure: `apply-templates` *and Mode*

The main control structure of ref2link.xsl is the `xsl:apply-templates` element. Listing 5-14 shows the beginning of our link source to linkbase transformer.

Listing 5-14. XLink Application: The Start Tag and the Root Template

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                version="1.0">
<xsl:output method="xml" indent="yes" encoding="iso-8859-1"
     doctype-public="elt" doctype-system="/xmlp/bibleLinks/biblelinkbase.dtd"/>
<!-- indent attribute specifies whether the XSLT processor may add
       additional whitespace when outputting the result tree; we also specify
      DOCTYPE elt PUBLIC /xmlp/bibleLinks/biblelinkbase.dtd -->
<xsl:param name="ref-dir" select="dat/jb/"> <!-- where ot.xml, nt.xml are kept -->
<xsl:template match="/">
  <elt xlink:type="extended">
    <xsl:apply-templates mode="locators"/>
    <xsl:apply-templates mode="arcs"/>
  </elt>
</xsl:template>
```

The top-level parameter `ref-dir` supplies the directory path to where source documents are kept. It is used in constructing the XPointer expressions later in the stylesheet.

The root template outputs the root element of the result document. It contains two `apply-templates` elements to process the source in two different modes.

Outputting Arcs

Although the stylesheet outputs locators first because such is the order of `apply-templates` elements, the actual order of templates in the stylesheet is immaterial. We have placed the arcs template first because it is much simpler. It uses two XPath string functions to compute the values of the `from` and `to` attributes. The values are inserted into the output using the curly-brackets notation for evaluating XSLT elements within attribute values, as explained in the section in Chapter 1, "Outputting Attributes with Computed Values".

```
<xsl:template match="ref" mode="arcs">
  <xsl:variable name="frombkchapverse" select="substring-after(from,' ')"/>
  <xsl:variable name="fromloc" select="translate($frombkchapverse,' ','_')" />
  <xsl:variable name="tobkchapverse" select="substring-after(to,' ')"/>
  <xsl:variable name="toloc" select="translate($tobkchapverse,' ','_')" />
  <elt xlink:type="arc" xlink:from="{$fromloc}" xlink:to="{$toloc}"/>
</xsl:template>
```

The `translate()` *Function*

The `substring-after()` function is self-explanatory, but `translate()` requires a comment. It takes three arguments: `string`, `targets`, and `replacements`, and replaces each character in `targets` with the corresponding character in `replacements`. If there are extra `targets`, they are deleted from the string; extra `replacements` are ignored. Here are some examples:

```
<xsl:variable name="s" select="axbxcx"/>
<xsl:value-of select="translate($s,'abc','zzz')"/>
result is: zxzxzx
<xsl:value-of select="translate($s,'ax','z')"/>
result is: zbc
<xsl:value-of select="translate($s,'a','zextra')"/>
result is: zxbxcx
```

Outputting Locators

For each verse mentioned in the `ref` elements of the source document, we must output a `locator` element, and we want to do it only once. Our task is to transform

```
<ref>
   <from>nt.xml Luke 18 27</from>
   <to>ot.xml Genesis 18 14</to>
 </ref>
into
   <elt
 xlink:label="Luke_18_27"
 xlink:href=
"dat/jb/nt.xml#xpointer(/tstmt/bookcoll/book[bktshort='Luke']/chapter[18]/v[27])"
 xlink:type="locator"/>
```

and a similar element for the other verse. The task of constructing the XPointer is implemented as a separate named template, which is discussed in the next subsection.

Listing 5-15. XLink Application: Outputting Locators

```
<xsl:template match="ref/to | ref/from" mode="locators">
  <xsl:if test="not(. = preceding::from | preceding::to)" >
      <!-- if not yet encountered -->
    <xsl:variable name="label"
```

```
     select="translate(substring-after(.,' '),' ','_')"
    /><!-- construct the label using substring-after and translate -->
    <xsl:variable name="ref">
       <xsl:call-template name="make-ref">
                   <!-- outsource XPointer production to make-ref -->
         <xsl:with-param name="doc-bk-chap-verse" select="." />
       </xsl:call-template>
    </xsl:variable>
    <elt xlink:type="locator"
     xlink:href="{$ref}" xlink:label="{$label}"
    /><!-- finally, literal result element with two computed attribute values -->
  </xsl:if>
</xsl:template>
```

The highlighted line checks to see whether the current node has yet been encountered in the input document. It uses the = operator to check whether the intersection of two node-sets is empty, as discussed in the section "Path Expressions" earlier in the chapter.

Constructing XPointers

The only remaining part of `ref2xlink` is the `make-ref` template. It breaks the input parameter into pieces using `substring-before()` and `substring-after()`, and constructs the required string out of those pieces using `concat()`.

Listing 5-16. XLink Application: Constructing XPointers

```
<xsl:template name="make-ref">
  <xsl:param name="doc-bk-chap-verse" select="''" />
<!-- break doc-bk-chap-verse into pieces: doc, bk, chap, verse -->
  <xsl:variable name="doc" select="substring-before($doc-bk-chap-verse,' ')"/>
  <xsl:variable name="bk-chap-verse"
                      select="substring-after($doc-bk-chap-verse,' ')"/>
  <xsl:variable name="bk" select="substring-before($bk-chap-verse,' ')"/>
  <xsl:variable name="chap-verse" select="substring-after($bk-chap-verse,' ')"/>
  <xsl:variable name="chap" select="substring-before($chap-verse,' ')"/>
  <xsl:variable name="verse" select="substring-after($chap-verse,' ')"/>
  <xsl:variable name="Q">'</xsl:variable>
<!-- construct the first part of URI+XPointer, up to book/verse/chapter -->
  <xsl:variable name="uri1"
    select="concat($ref-dir, $doc, '#xpointer(/tstmt/bookcoll/') "/>
<!-- construct the second part of URI+XPointer, up to chapter -->
```

```
  <xsl:variable name="uri2"
    select="concat('book[bktshort=', $Q, $bk,$Q,']/')" />
<!-- construct the third and last part of URI+XPointer -->
  <xsl:variable name="uri"
    select="concat("'chapter[', $chap, ']/v[', $verse, '])')" />
<xsl:value-of
   select="concat($uri1, $uri2, $uri3)" />
</xsl:template>
```

This concludes ref2xlink.xsl; neighbors.xsl is next. You may want to review Figure 5-1 that shows its operation.

Find-Neighbors Query

This program receives a verse reference on input and follows all the to-arcs that originate from that verse. (See Figure 5-1 for a screenshot.) To receive input, the stylesheet needs a top-level parameter, and to produce its output, the stylesheet has to URL encode some XPointers, exactly as we did in the Joseph Dreams XLink application of Chapter 2. As in Chapter 2, we call a Java extension function to do the encoding and therefore have to declare the appropriate namespace. Here is the part of the program that precedes the first template:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns:java="http://xml.apache.org/xslt/java"
                version="1.0">
<xsl:output method="html"/>
<xsl:param name="startVerse" select="''"/>
```

The rest of the program consists of four templates: the unnamed “root” template and three named ones: `make-row`, `make-rows`, and `nextlinks`.

The Root Template

The root template outputs the containing HTML tags: `html`, `body`, and `table`. Within the table, it calls three named templates:

- `make-row`: to fill in the first row of the table with the contents of startVerse
- `nextlinks`: to collect all arcs originating from startVerse into a variable
- `make-rows`: to fill in the rest of the table with the content of the end points of those arcs

Listing 5-17. Find-Neighbors: The Root Template

```
<xsl:template match="/"> <!-- apply to linkbase, e.g. biblelinkbase.xml -->
  <html><head>
    <title>Direct links from <xsl:value-of select="$startVerse"/></title>
  </head><body>
    <h4>Bible Verses directly linked from
      <xsl:value-of select="$startVerse"/>:</h4>
    <table border="1">
      <xsl:call-template name="make-row">
         <xsl:with-param name="verse-label" select="$startVerse"/>
      </xsl:call-template>
      <xsl:variable name="verses">
        <xsl:call-template name="nextlinks">
          <xsl:with-param name="verse" select="$startVerse"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:call-template name="make-rows">
        <xsl:with-param name="verses" select="$verses"/>
      </xsl:call-template>
    </table>
   </body></html>
</xsl:template>
```

The `make-row` *Template*

The `make-row` template is very similar to the last (unnamed) template in the XLink application of Chapter 2: it constructs an XPointer expression and uses it to retrieve data from an XML document, as shown in Listing 5-18.

Listing 5-18. Find-Neighbors: `make-row`

```
<xsl:template name="make-row">
  <xsl:param name="verse-label" select="''"/>
  <xsl:variable name="bk" select="substring-before($verse-label,'_')"/>
  <xsl:variable name="cv" select="substring-after($verse-label,'_')"/>
  <xsl:variable name="chapter" select="substring-before($cv,'_')"/>
  <xsl:variable name="verse" select="substring-after($cv,'_')"/>
  <xsl:variable name="h"
      select="//elt[@xlink:label=$verse-label]/@xlink:href"/>
  <xsl:if test="string-length($h)!=0">
    <xsl:variable name="x" select="substring-before($h,'#xpointer')"/>
    <xsl:variable name="p" select="substring-after($h,'#xpointer')"/>
```

```
        <xsl:variable name="x-enc" select="java:java.net.URLEncoder.encode($x)"/>
        <xsl:variable name="p-enc" select="java:java.net.URLEncoder.encode($p)"/>
        <xsl:variable name="qstring" select="concat('x=',$x-enc,'&p=',$p-enc)"/>
        <xsl:variable name="uri"
             select="'http://localhost:8080/xmlp/xptrans.jsp?'"/>
        <tr><td>
          <b><xsl:value-of select="concat($bk,' ',$chapter,':',$verse,' ')"/></b>
         <xsl:apply-templates select="document(concat($uri,$qstring))/*" />
        </td></tr>
   </xsl:if>
</xsl:template>
```

The first highlighted line searches for the `elt` element whose `xlink:label` attribute is equal to the label of the parameter to the template. As shown, this search is performed in the most inefficient manner and can be improved. For instance, if the document contained IDs, we could rewrite that line as

```
<xsl:variable name="h" select="id($verse-label)/@xlink:href"/>
```

We leave it as an exercise to restructure the data and the application to make this optimization possible. In the next chapter, we will show another possible optimization that also yields constant-time search but that does not require any changes in the data, only in the program.

The second highlighted line, as discussed in Chapter 2, sends the data extracted from an external document looking for templates that would process it.

The `nextlinks` *Template*

The `nextlinks` template creates a space-separated and space-terminated list of tokens. Each token is the value of the `xlink:to` attribute in an arc whose `xlink:from` attribute is our startVerse. As you may recall from our XLink discussion in Chapter 2, the values of those attributes are string tokens. The list of those tokens becomes the value of the `verses` variable in the root template and gets passed to `make-rows`, as shown in Listing 5-19.

Listing 5-19. Find-Neighbors: `nextlinks`

```
<xsl:template name="nextlinks">
  <xsl:param name="verse" select="''"/>
  <!-- produces links directly reachable from current link -->
  <xsl:for-each select="//elt[$verse=@xlink:from]/@xlink:to" >
     <xsl:value-of select="concat(.,' ')"/>
```

```
      <!-- add a space to each, including the last one -->
  </xsl:for-each>
</xsl:template>
```

The make-rows *Template*

The make-rows template outputs the rest of the table. It uses the same linear-recursion pattern as the make-rows template of Listing 5-13 that worked on persons and their helpers. (See Listing 5-20.)

Listing 5-20. Find-Neighbors: make-rows

```
<xsl:template name="make-rows">
  <xsl:param name="verses" select="''"/>
  <xsl:if test="string-length($verses)!=0"><!-- until "$verses" is empty -->
     <xsl:call-template name="make-row">
       <xsl:with-param name="verse-label" select="substring-before($verses,' ')"/>
     </xsl:call-template>
     <xsl:call-template name="make-rows">
       <xsl:with-param name="verses" select="substring-after($verses,' ')"/>
     </xsl:call-template>
  </xsl:if>
</xsl:template>
```

This completes the find-neighbors query. In the next section, we generalize it to produce reachable.xsl, the transitive closure of one-step search.

The All-Reachable Query

In abstract algorithmic terms, our task in this section is to search a directed graph for all nodes that are reachable from a given node. The standard procedure is as follows:

1. Form a list of nodes to visit, initially containing only the start node.

2. Repeat until that list is empty. Visit the first node on the list unless it has already been visited, and add its neighbors to the list.

The program maintains two lists: visited nodes (initially empty) and nodes-to-visit (which initially contains only the start node). If the list of nodes to visit is maintained as a stack (latest additions go to the front), the graph is traversed in

"depth-first" order: we go as deeply as we can following the "first-neighbor" links before we try the second neighbor of the original node. If the list is maintained as a queue (latest additions go to the back), the graph is traversed in "breadth-first" order. The code for the two traversals is almost identical, differing only in a single line that adds new neighbors to the list.

> **NOTE** *The book's code contains a "decorated" version of the program, reachable-dist.xsl, that helps visualize the traversal process. It outputs each node together with its distance from the start node. In the breadth-first traversal, all nodes of distance 1 are displayed before all nodes of distance 2, and so on. In the depth-first traversal, distance increases until we hit a dead end and backtrack to a preceding distance. In addition, the program outputs messages (*`xsl:message` *elements) showing the contents of the nodes-to-visit list and the visited-nodes list at every stage of the traversal. We recommend that you run this program to see how it unfolds.*

In Listing 5-21, we use the `$queue` parameter to hold the list of nodes to visit and the `$verses` parameter to hold the list of visited nodes. Both are implemented as a space-separated and space-terminated string, so we can perform the familiar recursion. The program reuses a good deal of material from the preceding section: the `make-row` and `make-rows` templates are identical, but the root and `nextlinks` templates have minor changes. The main addition is the `closure` template that gets called from the root template. The `closure` template's control structure is as follows:

- If the list of nodes to visit is empty, the closure template sends `$verses` to the familiar `make-rows` template that sends them to output.
- If that list is not empty, it is divided into two parts: the first node and the tail end.
- If the first node has already been visited, the closure template calls itself recursively with the tail end of `$queue` as parameter.
- If the first node has not yet been visited, it is added to `$verses`, and its neighbors are added to `$queue`. With the parameters so updated, closure recursively calls itself.

Because Listing 5-21 is fairly long, we highlight a few important comments and lines showing main divisions in the code.

Listing 5-21. All-Reachable: The Root and `closure` *Templates*

```
  <xsl:template match="/"> <!-- apply to linkbase, e.g. biblelinkbase.xml -->
  <html><head>
    <title>Reachable from <xsl:value-of select="$startVerse"/></title>
  </head><body>
    <h4>Bible Verses reachable from <xsl:value-of select="$startVerse"/>:</h4>
    <table border="1">
      <xsl:call-template name="closure">
        <xsl:with-param name="queue" select="concat($startVerse,' ')" />
        <xsl:with-param name="verses" select="''" />
             <!-- redundant; added for clarity -->
      </xsl:call-template>
    </table></body></html>
</xsl:template>
<xsl:template name="closure">
   <xsl:param name="verses" select="''"/>
   <xsl:param name="queue" select="''"/>
   <xsl:choose>
     <xsl:when test="not($queue)">
          <!-- queue is empty; we're done; produce output -->
       <xsl:call-template name="make-rows">
          <xsl:with-param name="verses" select="$verses"/>
       </xsl:call-template>
     </xsl:when>
     <xsl:otherwise>
       <xsl:variable name="verse" select="substring-before($queue,' ')"/>
                 <!-- first node -->
       <xsl:variable name="tl-queue" select="substring-after($queue,' ')"/>
            <!-- tail end -->
       <xsl:choose>
         <xsl:when test="contains($verses,concat($verse,' '))">
           <!-- first node already visited -->
            <xsl:call-template name="closure"><!-- recursive call -->
              <xsl:with-param name="verses" select="$verses"/>
              <xsl:with-param name="queue" select="$tl-queue"/>
            </xsl:call-template>
         </xsl:when>
         <xsl:otherwise>
            <xsl:variable name="new-links"><!-- get neighbors of first node -->
               <xsl:call-template name="nextlinks">
                 <xsl:with-param name="verse" select="$verse"/>
               </xsl:call-template>
            </xsl:variable>
```

```
          <xsl:call-template name="closure"><!-- recursive call -->
            <xsl:with-param name="verses"
                                  select="concat($verses,$verse,' ')"/>
            <!-- next line controls order of traversal:
                  depth-first vs. breadth-first -->
            <xsl:with-param name="queue"
                                 select="concat($new-links,$tl-queue)"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

The rest of the program consists of three templates: `make-rows` (identical to Listing 5-20), `make-row` (identical to Listing 5-18) and `nextlinks`. The `nextlinks` template is almost identical to Listing 5-19 except it now compares `$verse` to both `from` and `to` links, as shown by the two highlighted lines:

```
<xsl:template name="nextlinks">
  <xsl:param name="verse" select="''"/>
      <xsl:for-each select=
             "//elt[$verse=@xlink:from]/@xlink:to |
               //elt[$verse=@xlink:to]/@xlink:from" >
        <xsl:value-of select="concat(.,' ')"/>
      </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

This concludes our discussion of reachable.xsl. Parts of it can be made more efficient by using the `xsl:key` element, which is the subject of the next chapter.

Conclusion

XSLT is a powerful technology that should be in every XML programmer's toolbox. In this chapter, we covered XSLT in considerable detail. We did not aim to be comprehensive, but instead tried to identify the most-important and commonly used features of the language and common patterns of use, with an emphasis on efficiency. We also developed an XSLT program to work with XLink structures.

XSLT heavily relies on XPath for much of its operation. XPath is another language that is at the core of XML processing, and, in addition to XSLT, it forms an essential part of XPointer and XQuery. These technologies are not yet standard, but they are developing rapidly.

All the themes of this chapter are continued in the next one, where we'll spend more time on issues of efficiency. We will show several common advanced uses of XSLT and the "best practice" approaches to them, and we will continue developing our XLink application, both to add functionality and improve the efficiency of its code.

CHAPTER 6

More XSLT: Algorithms and Efficiency

THIS CHAPTER DEVELOPS SEVERAL TOPICS, all of them having to do with writing more efficient programs and measuring a program's efficiency. A program's efficiency can be improved by using a more suitable data structure, a better algorithm, or both. XSLT offers an efficient data structure for retrieval through its `xsl:key` element and the accompanying XPath `key()` function. Their combined effect is to improve efficiency at runtime (when the transformations are performed) by doing some preprocessing at "parse time" when the XML input is parsed into the XPath tree. The preprocessing at parse time is triggered and controlled by `xsl:key`, and search at runtime is performed by the `key()` function. We explain the workings of `xsl:key` in some detail and show how it can be used in several common tasks:

- Create a list of distinct values for a given element, such as a list of all speakers in a play or all affiliates of a company mentioned in a financial report.
- Group elements together by some predicates: for instance, group sales records by geographic region.
- Convert flat XML structures, such as those extracted from a relational database or an Excel spreadsheet, into more-deeply nested structures that reflect semantic groupings of data.

Because of the importance of recursion in XML programming, we also show how to improve the efficiency of recursive programs. In particular, recursive programs often fail because the processor runs out of stack memory. We show how a more elaborate algorithm can process much larger data sets than the more obvious uses of recursion that you have seen in the preceding chapter. The two problems we use to illustrate these kinds of issues are that of generating a large range of integers and finding the maximum-length element in a large data set.

Another well-known way to improve efficiency is to offload a resource-intensive subroutine to a more efficient (and usually less portable) language processor: code a C function in assembler or a LISP function in C. XSLT offers

a mechanism of extension functions that allow some tasks to be offloaded to a function written in Java or a common scripting language, such as JavaScript and VBScript. Although not yet standardized, the mechanism is offered by all the major XSLT processors and will become standard in *XSLT 2.0*.

In outline, the chapter proceeds as follows:

- optimization by using more specific patterns
- use of `xsl:key` and `generate-id()` to create a list of distinct values
- use of distinct values for grouping and tabulation
- use of patterns in flat data to add more structure
- patterns of list processing
- linear recursion vs. tree recursion and the use of stack memory
- generating large data sets
- use of extension functions for improved performance

We start with a simple idea: specific paths take less time to process than do patterns that use the `//` operator. We use a JavaScript utility to confirm that the idea is valid and to measure possible gains.

Specific Patterns and Timing

In this section, we show how you can make your program more efficient by using specific patterns and how you can measure the gain in efficiency. We will choose the simple task of counting the number of times that a specific element occurs in a document, such as how many distinct speeches are in a Shakespeare play (marked up by Jon Bosak according to his `play.dtd`). A simple and the least efficient way to do that is

```
<xsl:value-of select="count(//SPEECH)"
```

Inspecting the DTD, we easily discover that a `SPEECH` element can occur in exactly seven different contexts, corresponding to seven different path expressions. Therefore, we can rewrite this line as

```
<xsl:value-of select="count(/PLAY/INDUCT/SCENE/SPEECH)+
                      count(/PLAY/INDUCT/SPEECH)+
                      count(/PLAY/PROLOGUE/SPEECH)+
                      count(/PLAY/ACT/PROLOGUE/SPEECH)+
                      count(/PLAY/ACT/SCENE/SPEECH)+
                      count(/PLAY/ACT/EPILOGUE/SPEECH)+
                      count(/PLAY/EPILOGUE/SPEECH)"/>
```

Intuitively, the second expression must take less time to evaluate because we tell the processor exactly where `SPEECH` elements are to be found, instead of sending it all over the tree looking for them. How do we confirm this intuition and get an estimate on the kind of savings we can achieve?

Timing a Web Application

One possible approach is to run the XSLT processor as a Web application and ask JavaScript for timestamps right before and right after. Here is an implementation of this idea, in the file timeXSLT\timeXSLTctl.htm. Listing 6-1 is the "control frame," named `ctlFrame`, of a two-frame document, timeXSLT/timeXSL.htm.

Listing 6-1. Timing a Stylesheet

```
<html><head><title>Timing Speech Counts</title>
<script type="text/javascript" language="javascript">
  function callBack(){
    with(theForm){
// endTime, startTime and totalTime are fields in the form called theForm,
// in the body of the document below
      endTime.value=new Date().getTime();
      totalTime.value=endTime.value - startTime.value;
    }
   }
  function onSubmitForm(){
    theForm.startTime.value=new Date().getTime();
    return true;
  }
</script></head><body>
<form name="theForm" action="/xmlp/xx.jsp"
  onSubmit="return onSubmitForm()" target="dataFrame" method="post">
```

```
xmlUri<input type="text" name="xmlUri" value="/dat/jb/macbeth.xml" /><br/>
xslUri<input type="text" name="xslUri" value="/timeXSLT/countspeakers.xsl"/><br/>
startTime<input type="text" name="startTime" value=""/><br/>
endTime<input type="text" name="endTime" value=""/><br/>
totalTime<input type="text" name="totalTime" value=""/><br/>
<input type="submit" value="GetTiming"/>
</form></body></html>
```

The output of `onSubmitForm()` goes into the "data frame," initially empty. That output is generated by xx.jsp that runs an XSLT program on an XML document. (You saw it in Chapters 1 and 2.) To make the timing code work, the XSLT outputs a call on the JavaScript `callBack()`, as shown by the highlighted line in Listing 6-2.

Listing 6-2. XSLT Stylesheet with a JavaScript Callback

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
  <html> <head> <title> CountSpeeches </title> </head>
    <body onload="parent.ctlFrame.callBack()">
     <p>Count Speeches in a play, wherever they occur.</p>
     <p>
     The total number of speeches is <br/>
     <xsl:value-of select="count(//SPEECH) "/>
     </p></body></html>
 </xsl:template></xsl:stylesheet>
```

The durations recorded by JavaScript include, in addition to actual XSLT processing, the time required to parse the XML source, parse the stylesheet, and generate the output, not to mention the overhead of the Tomcat process and IE. It is all the more remarkable that, in running the two versions of the stylesheet on the macbeth.xml file, we originally found the less specific `//` template (timeXSLT/countspeeches.xsl) taking almost twice the amount of time of the more specific one (timeXSLT/countspeechespull.xsl): from a range of 1,800 to 2,000 milliseconds to a range of 800 to 1,100 milliseconds. (This was on a Dell Dimension 450 running Windows 98, Sun's JDK 1.2.2, and Xalan 2.0 with Xerces 1.4.1 under Tomcat 3.2.) Your times will of course be faster: even on the same machine, they improve substantially running JDK 1.3 with Xalan 2.2.0 and Xerces 1.4.4 under Tomcat 4.0.1. Noise comes to dominate: as the numbers shrink, the variation in the time becomes as large as the time itself, and we have to perform experiments with larger documents.

The use of `//` is frequently considered a sign of inadequate thinking and disapproved of. We consider it to be a sign of a rough draft, something which can

almost always be improved but which is quite useful in the early stages of development.

Distinct Nodes and Keys

Our next problem will be to find and in some way process only distinct elements that may repeat more than once in the input document. We did a problem like that in the ref2xlink.xsl stylesheet of Chapter 5 (Listing 5-14). It creates a list of distinct locators out of `ref/to` and `ref/from` elements in the linkbase source document:

```
<xsl:template match="ref/to | ref/from" mode="locators">
<!-- if an element is not yet encountered -->
  <xsl:if test="not(. = preceding::from | preceding::to)" >
<!-- construct xlink:href and xlink:label attributes
       as ref and label variables -->
    <elt xlink:type="locator" xlink:href="{$ref}" xlink:label="{$label}" />
  </xsl:if>
</xsl:template>
```

As explained in Chapter 5, the highlighted line checks to see whether the current element occurs among preceding elements. Because we do the check for each `from|to` element on input, and each check requires a linear search through all preceding such elements, the entire time of the procedure is proportional to the square of the number of `from|to` elements. We would like to reduce this to linear time, by arranging to do the search in constant time.

In working with people.xml, we were able to reduce linear search to constant-time search by using the `id()` function. (We had to rely on a partial DTD that declared an attribute of type ID for the `person` element.) This time around, we do not have this option because both speakers in Shakespeare plays and `to|from` elements in the linkbase source repeat many times in different contexts in the source document and cannot be fixed with an ID (even if we had license to change the source document). We will have to use a different technique, using the `xsl:key` element and the `key()` function.

The `xsl:key` *Element and the* `key()` *Function*

`xsl:key` is a top-level element. It is always empty but has three required attributes: `name`, `match`, and `use`. As often in XSLT, the complete description of `xsl:key` is rather involved but the basic use is straightforward. Think of it as defining

a named hashtable of nodes within a document keyed by the values of an XPath expression. For instance, with people.xml as input, we can create a key like this:

```
<xsl:key name="person" match="person" use="@id"/>
```

This creates—at parse time—a table of persons keyed by their `id` attributes. More precisely:

- The name of the hashtable is the value of the `name` attribute.
- The nodes stored in the hashtable are those that match the `match` pattern.
- The key for a given node is the value of the `use` attribute evaluated as an XPath expression, with the node as the current node.

The accompanying `key()` function takes two arguments: the name of the hashtable and a key value. The function returns the node-set of all nodes that are keyed by the given value. To retrieve the first (and in this case unique) person whose ID is "A4", we would say

```
key("person","A4")[1]
```

This is likely to take constant time and does not require attributes of type ID declared in the DTD. As often the case, we save time by using extra (memory) space: the key is created and kept in memory. It may or may not be implemented as a hashtable, but we find it a useful conceptual prop to think that it is. Implementers are certainly aware that XSLT programmers depend on this function for speed.

To repeat the main point: the purpose of `xsl:key` and the accompanying `key()` function is to offload to parse time some preprocessing that results in efficient (constant-time) search at runtime. The preprocessing at parse time is triggered and controlled by `xsl:key`, and search at runtime is performed by the `key()` function.

Distinct Locators

The `key()` function returns a node-set that may contain any number of nodes. For instance, with Bible references, we can define a key like this:

```
<xsl:key name="distinct" match="ref/to|ref/from" use="."/>
```

This says “put all `ref/to` and `ref/from` nodes into the table and key them by their own string value.” In other words, a `ref/from` or `ref/to` element that contains the string “nt.xml Luke 18 27” will be keyed by that string. With the table so constructed, each key value is associated with a node-set that can contain more than one node because the same verse in the Bible may appear more than once.

How do we extract distinct locators from such a table? The crucial support comes from the `generate-id()` function (an XPath function defined within XSLT). For each node in a document, the function generates a string value that is guaranteed to be unique within that document. (Different processors will produce different such values, depending on the algorithm they use.) Using `generate-id()`, we can extract the first item of each node-set stored in the `distinct` key. They will all be distinct, and together they will contain all the distinct items. Listing 6-3 shows the code of the timeXSLT/distinctrefs.xsl file, to be applied to biblelinks/biblelinks.xml.

Listing 6-3. Unique Locators Using `xsl:key` *and* `generate-id()`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html" />
<xsl:key name="distinct" match="ref/to | ref/from" use="." />
<xsl:template match="/">
  <html> <head> <title>UniqueSpeakers </title> </head><body>
    <xsl:apply-templates select="*/ref/to | */ref/from" />
      <!--collect all to|from nodes -->
  </body></html>
</xsl:template>
<xsl:template match="ref/to | ref/from">
 <xsl:if test="generate-id(.)=generate-id(key('distinct',.)[1])">
   <p><xsl:value-of select="."/></p>
 </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

NOTE *Strictly speaking, the* `[1]` *predicate is unnecessary because, if* `generate-id()`*’s argument is a node-set that contains more than one node, the function generates a unique string based on its first node. If we want to compare a node N with the first node in a node-set NS, we can simply say* `generate-id(N)=generate-id(NS)`*. However, many examples in the literature contain the predicate (perhaps to avoid having to give this explanation), and we will follow their practice.*

The test takes constant time, and so the entire template takes linear time but it's linear in the number of `to|from` elements in input data. These elements get collected into a node-set at runtime, by the `xsl:apply-templates` expression. If the tree is large, we can further improve efficiency by creating that node-set at parse time, as another key that will hold all `to|from` nodes. The value of its `use` attribute will simply be a constant string, and the entire table will hold a single node-set in the timeXSLT/distinctrefs2keys.xsl file:

```
<xsl:key name="distinct" match="ref/to | ref/from" use="." />
<xsl:key name="keylist" match="ref/to | ref/from" use="keyVal" />
<xsl:template match="/">
  <html> <head> <title>Unique Locators</title> </head><body>
    <xsl:apply-templates select="key('keylist','keyVal')" />
  </body></html>
</xsl:template>
<!--continue as before -->
```

As an exercise, try using the same techniques for generating a list of speakers in a Shakespeare play.

The Grouping Problem

The idea of using `generate-id()` in combination with `xsl:key` to produce a list of all possible values of some property without repetition was first discovered (by Steve Muench of Oracle) in the context of "the grouping problem." The grouping problem is to divide a data set into groups using the values of some property, such as group sales figures by geographic region. Databases and spreadsheets do grouping efficiently; in XSLT, the task frequently arises in working with XML output from databases.

Staying with the same material, suppose that we have a large collection of `from|to` links and we want to group them by to-locators to answer the query: "for each locator, show all links that point to it." Two sub problems must be solved here:

- Create a list of locators, without repetition, as in the preceding section.
- For each locator, find its to-links.

We have, in effect, two nested loops: the outer loop produces a list of distinct to-locators, and the inner loop outputs the corresponding from-locators. Listing 6-4 is the code of groupXSLT/groupref-to.xsl.

Listing 6-4. Grouping References by To-Locators Using `xsl:key` *and* `generate-id()`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html" />
<xsl:key name="to-locs" match="ref/to" use="." />
<xsl:template match="/">
  <html> <head> <title>References grouped by to-locator</title> </head><body>
<xsl:for-each select="*/ref/to[generate-id()=generate-id(key('to-locs',.)[1])] ">
   <h1><xsl:value-of select="."/></h1>
   <xsl:for-each select="key('to-locs',.)">
     <xsl:text>from: </xsl:text>
     <xsl:value-of select="../from"/>
     <br/>
    </xsl:for-each>
</xsl:for-each>
  </body></html>
</xsl:template>
</xsl:stylesheet>
```

As you can see, the outer loop closely follows the pattern of Listing 6-3. The complementary problem that groups by from-locators is easy; try to solve it before looking at the code for groupXSLT/groupref-from.xsl.

Grouping and Tables

In the examples of the preceding section, records are grouped by some element of their content. In this section, we go over a few cases in which records need to be broken into groups such that each group contains the same number of items. An essential tool is the XPath `mod` operator that produces the remainder of integer division (technically "the remainder from a truncating division"). Its operation with negative numbers is not quite obvious (see the W3C XPath recommendation), but you will rarely if ever use it with negative numbers. With positive numbers, `mod` is useful for dividing items into groups of equal size: if your variable `nodeSetVar` has a node-set value, then the expression

```
$nodeSetVar[position() mod 5=1]
```

will output every fifth node in the node-set beginning with the first.

Outputting a Table

Suppose that you have twenty items and you want to output them as a table with four rows and five columns. A possible strategy is to collect the items that will end up in the first column of such a table into a node-set, and for each item in that node-set produce a row. Assuming that the number of columns is in parameter `numColumns`, we can construct a table as follows:

```
<xsl:for-each select="xpath/to/data[position() mod $numColumns = 1]">
   <tr>
       <td><xsl:value-of select="."/></td>
          <xsl:for-each
              select="following-sibling::*[position() &lt; $numColumns]">
       <td><xsl:value-of select="."/></td>
   </tr>
</xsl:for-each>
```

This makes several assumptions: that the data is already in row-first order, ready to be tabulated; that all rows are intended to be of equal length; and that the data set is small enough to be processed in reasonable time. In the remainder of this section, we'll work on problems in which at least one of these assumptions doesn't hold.

Table Regrouping with Summation by Category

Suppose that our data items have two properties and need to be grouped into rows and columns by the values of those properties rather than in the order given. A common example would be company sales data that is initially presented by division, and within each division monthly sales totals of the regions covered by that division. What would you do to find the aggregate monthly sales for each region, ignoring the divisions? In other words, we want to perform regrouping and accumulations within the groups. Given what we already know, a two-step strategy suggests itself. First, collect each "row category" into a key, using the "Muenchian" technique. Second, for each row, output the values of the "column category" extracting them from the appropriate key.

With this strategy in place, we need some data to try it on. In this example, we will start with an Excel table, save it as HTML, convert HTML to XHTML, convert XTHML to another XML vocabulary using table headers for tag names, and finally restructure it into a different table by grouping together items by category.

> **NOTE** *HTML-to-XHTML conversion is best done by Tidy, a program by Dave Raggett of W3C that we mentioned in Chapter 2. For information on how to get Tidy, see Appendix D.*

From Excel to HTML

The following sample (Figure 6-1) of initial Excel data, groupXSLT/DivReg.xls, shows monthly sales data by region and division. For some months, some divisions generated no sales in some regions.

Microsoft Excel - DivReg1.xls

	A	B	C	D	E	F	G
1				ABCD Company			
2							
3			Reg 1	Reg 2	Reg 3	Reg 4	Reg 5
4	Div A						
5		January	300457	277753	488524	282375	280358
6		February	425802	451557	214516	143990	406442
7		March	301631	479363		273198	131533
8		April	291340	213482	326823	389354	266643
9		May	180244	255503	473330	161774	491852
10		June	105163	317281	383406	171133	421514
11		July	268072	176172	307229	286144	487385
12		August	429488	125521	150661	232732	272981
13		Septembei	457896	205353	463265	421544	232923
14		October	484243		154632	107246	145201
15		November	424810	235285	105422	107078	140884
16		December	381596	184166	404491	174882	298024
17	Div B						
18		January	366420	349606	434484	175700	275949
19		February	454190	227428	156658	332234	
20		March	357803	140013	280221	438754	461497
21		April	164298	333375	337710	222528	473530
22		May	246491	398143	118797	469325	491476

Figure 6-1. Initial Excel data

To convert this data to HTML (Office 2000 version because Office 1997 works differently), select the range of data to save and choose “Save as Web Page” from the File menu. The resulting HTML is mostly a table with many empty cells. To

give an example, row 6 of the table (Div A, February), looks like this (with some whitespace removed):

```
<tr>
  <td></td>
  <td>February</td>
  <td align=right x:num>425802</td>
  <td align=right x:num>451557</td>
  <td align=right x:num>214516</td>
  <td align=right x:num>143990</td>
  <td align=right x:num>406442</td>
 </tr>
```

Several items in the output make it non-well-formed: empty elements without closing tags, unquoted attribute values, improperly formatted comments, and so on. Before XLST can work on this data, it must be converted to well-formed XHTML.

Tidy Use

HTML-to-XHTML conversion is done by Tidy. Depending on option settings, Tidy will output fully conformant HTML or XHTML or generic XML; for HTML and XHTML, it may use the strict or transitional DTD (that is, it may or may not convert `<center>..</center>` into `<div align = "center">..</div>`). Our settings are collected in the ciacfg.txt file. To produce correct results on Excel-generated data, the settings must include this line:

```
word-2000:yes
```

Tidy's output, in groupXSLT/DivReg.xml, is ready for XSLT.

From XHTML to XML

Our next task is straight XML restructuring, converting the XHTML table structure into

```
<salesData>
<item div="Div A" reg="Reg 1" mo="January">300457</item>
<!-- many more items; skip empty cells -->
</salesData>
```

The peculiarity of our data is that we have three different kinds of rows:

- the top row that lists region names ("the regions row")
- the first row of each division that shows the division's name but otherwise has no data ("division rows")
- the remaining rows that have no data in the first `<td>` ("data rows")

Because we have to provide the division name for each item on the output, we collect all data rows into a key and index each data row by the first preceding division row. As you read the code of groupXSLT/DivReg.xsl, remember that input XML is, in fact, XHTML, defined to be in the `http://www.w3.org/TR/REC-html40` namespace.

Because the namespace is declared on the root element, all other elements in the stylesheet inherit it, as required by *Namespaces in XML* recommendation. The XPath tree will represent it as multiple namespace nodes attached to element nodes. The XSLT processor, unless told otherwise, will copy all these irrelevant namespace nodes to output. The `exclude-result-prefixes` attribute prevents this from happening; you will see it used in many examples of this chapter:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:h="http://www.w3.org/TR/REC-html40"
  exclude-result-prefixes="h">
<xsl:output method="xml" indent="yes"/>
<xsl:strip-space elements="*"/>
<xsl:key name="divrows" match="h:tr[not(h:td[1]/node())]"
   use="generate-id(preceding-sibling::h:tr[h:td[1]/node()][1])"/>
<!-- for each non-header row, h:td[1] is empty, and so not(h:td[1]/node())
  is true. The most recent row such that h:td[1]/node() is true is the
  most recent header row.
-->
```

With the `divrows` key so defined, for any header row, `key('divrows',generate-id(.))` is the list of rows for which it is header. We are now ready to process data rows, beginning in row 4 (so we specify `position > 3`). Here is the rest of groupXSLT/DivReg.xsl:

```
<xsl:template match="/">
<xsl:apply-templates select="/h:html/h:body"/>
</xsl:template>
```

```
<xsl:template match="h:table">
  <salesData>  <!-- apply templates to header rows -->
  <xsl:apply-templates select="h:tr[position()&gt; 3][h:td[1]/node()]"/>
  </salesData>
</xsl:template>

<xsl:template match="h:tr"> <!-- we are in a header row -->
  <xsl:variable name="div" select="string(h:td[1])"/>
  <xsl:variable name="regions" select="../h:tr[3]/h:td[position()&gt;1]"/>
  <!-- outer loop: for each division row -->
  <xsl:for-each select="key('divrows',generate-id(.))">
    <xsl:variable name="mo" select="string(h:td[2])"/>
  <!-- inner loop: for each region listed in the regions row -->
    <xsl:for-each select="h:td[position()&gt;2]">
      <xsl:variable name="pos" select="position()"/>
      <xsl:if test="node()"><!-- skip empty cells -->
       <item div="{$div}" mo="{$mo}" reg="{$regions[$pos]}">
          <xsl:value-of select="."/>
       </item>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template></xsl:stylesheet>
```

From Data Items to Summary Table

This last transformation can be done very efficiently and elegantly with keys. No special provisions are needed for missing items (initially empty cells in the Excel table) because each item contains complete information about itself and is placed in the output table on the basis of that information.

First, we collect all items in two keys, indexed by the `reg` and `mo` attributes:

```
<xsl:key name="reg" match="item" use="@reg"/>
<xsl:key name="mo" match="item" use="@mo"/>
```

We also collect them in a key indexed by the combined value of those two attributes:

```
<xsl:key name="reg-mo" match="item" use="concat(@reg,' ',@mo)"/>
```

Even though `reg-mo` is a single key that keys each item by a single string, it is functionally equivalent to a two-dimensional table in which each item is keyed

by two values: the region and the month. Put differently, it captures triples of data: region, month, and item. Given a node-set of distinct regions and a node-set of distinct months, we can run two nested `xsl:for-each` loops over them and extract groups of items by a unique combination (concatenation) of the region and the month. We obtain the node-sets using the familiar technique:

```
<xsl:template match="salesData">
<xsl:variable name="regs"
    select="item[generate-id(.)=generate-id(key('reg',@reg))]"/
<xsl:variable name="mos"
    select="item[generate-id(.)=generate-id(key('mo',@mo))]"/>
```

At this point, we are ready to produce the output. Listing 6-5 shows the stylesheet groupXSLT/DivRegMonth.xsl, all together except for the root element and comments.

Listing 6-5. Table Output from Keys

```
<xsl:output method="xml" indent="yes"/>
<xsl:strip-space elements="*"/>
<xsl:key name="reg" match="item" use="@reg"/>
<xsl:key name="mo" match="item" use="@mo"/>
<xsl:key name="reg-mo" match="item" use="concat(@reg,' ',@mo)"/>
<xsl:template match="salesData">
<xsl:variable name="regs"
    select="item[generate-id(.)=generate-id(key('reg',@reg))]"/>
<xsl:variable name="mos"
    select="item[generate-id(.)=generate-id(key('mo',@mo))]"/>
<table border="1">
  <tr><th/><!-- a row of headers listing months -->
    <xsl:for-each select="$mos">
      <th><xsl:value-of select="@mo"/></th>
    </xsl:for-each>
  </tr>
  <xsl:for-each select="$regs"><!-- outside loop, a row per region -->
    <xsl:variable name="reg" select="@reg"/>
    <tr><td><xsl:value-of select="$reg"/></td>
      <xsl:for-each select="$mos"><!-- inside loop, add up region-month data -->
        <xsl:variable name="mo" select="@mo"/>
        <td>
            <xsl:value-of select="sum(key('reg-mo',concat($reg,' ',$mo)))"/>
        </td>
      </xsl:for-each>
    </tr>
  </xsl:for-each>
</table></xsl:template></xsl:stylesheet>
```

Because all the data is in keys, this program can, in effect, ignore its input and use the `key()` function for all its data needs.

U.S. Presidents by Quarter-Century and Party Affiliation

This section is a variation on the preceding one and leaves part of its code out as an exercise. We again proceed from XHTML to XML data to regrouped and summarized XML data. The difference is that we are using real data, and our categories are computed values rather than labels in the data itself.

From XHTML to XML

Our XHTML data (groupXSLT/presTable.htm) shows, for each past U.S. president, his years in office and party affiliation:

```
<table>
<tr><th>Name</th><th>From</th><th>To</th><th>Party</th></tr>
<tr><td>George Washington</td><td>1789</td><td>1797</td><td>None</td></tr>
<tr><td>John Adams</td><td>1797</td><td>1801</td><td>Federalist</td></tr>
<tr><td>Thomas Jefferson</td><td>1801</td><td>1809</td><td>Dem-Rep</td></tr>
...
<tr><td>William J. Clinton</td><td>1993</td><td>2001</td><td>Dem</td></tr>
</table>
```

For meaningful grouping, we need meaningful tag names. We transform the XHTML into a vocabulary extracted from table headers:

```
<?xml version="1.0" encoding="UTF-8"?>
<presidents>
  <president>
    <Name>George Washington</Name>
    <From>1789</From>
    <To>1797</To>
    <Party>None</Party>
  </president>
...
</presidents>
```

This is a straightforward transformation that we leave as an exercise.

From XML to XHTML Summary Table

Our next task is regrouping: we are going to generate a table of output in which each row is a quarter-century, each column is a political party, and each entry is the total number of presidents of that political party whose term began within that quarter-century. (See Figure 6-2.) This is done by PresSum.xsl; the result is in PresSumOut.htm.

http://localhost:8080/xmlp/xx.jsp?xmlUri=groupXSLT/presidents.xml&xslUri=groupXSLT/presSum.xsl&method=html

	None	Federalist	Dem-Rep	Dem	Whig	Rep
1775	1	1	0	0	0	0
1800	0	0	3	0	0	0
1825	0	0	1	3	3	0
1850	0	0	0	3	1	2
1875	0	0	0	2	0	5
1900	0	0	0	1	0	4
1925	0	0	0	2	0	1
1950	0	0	0	2	0	3
1975	0	0	0	2	0	2

Figure 6-2. Presidents by quarter-century and party affiliation

As did several earlier examples, PresSum.xsl starts by defining keys. To key each year by the quarter-century in which it belongs, we do integer division. (In XPath, this comes out as `floor(. div 25)` because there is no primitive integer division operation.)

```
<xsl:key name="party" match="Party" use="."/>
<xsl:key name="qcent" match="From" use="floor(. div 25)"/>
<xsl:key name="p-q" match="president"
              use="concat(Party,' ',floor(From div 25))" />
```

We create a list of parties and a list of quarter-centuries as before. To create a list of quarter-centuries, we locate, within each, the first year of a new presidency. (In the following code, the values of select attributes are broken into two lines for formatting.)

```
<xsl:variable name="parties"
    select="/presidents/president/Party
                 [generate-id(.)=generate-id(key('party',.))]"/>
<xsl:variable name="qcents" select="/presidents/president/From
                 [generate-id(.)=generate-id(key('qcent',floor(. div 25)))]"/>
```

The remaining processing (shown in Listing 6-6) is quite similar to Listing 6-5.

Listing 6-6. Presidents Grouped by Party and Quarter-Century

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>
<!-- create three keys as shown above -->
<xsl:template match="/">
<!-- create two variables as shown above -->
<table>
<tr><!-- output row of headers, skip first column, use party names -->
   <th></th>
    <xsl:for-each select="$parties">
      <th><xsl:value-of select="."/></th>
    </xsl:for-each>
</tr>
<xsl:for-each select="$qcents"><!-- output the rest of table -->
  <xsl:variable name="qc" select="floor(. div 25)"/>
  <tr><td><xsl:value-of select="25 * $qc"/></td>
    <!-- use quarter century for row label -->
    <xsl:for-each select="$parties"><!-- inner loop: output row -->
       <xsl:variable name="p" select="."/>
       <td><xsl:value-of select="count(key('p-q',concat($p,' ',$qc)))"/></td>
    </xsl:for-each>
  </tr>
</xsl:for-each></table></xsl:template></xsl:stylesheet>
```

The programs of Listing 6-5 and 6-6 are linear in the size of output, which is the best one can hope to achieve. (Output has to be produced, no matter what.) The efficiency is the result of a lot of preprocessing that's done at parse time and stored in efficient memory structures. This strategy works as long as keys can express relationships in data that we need to capture. It can also be applied to another class of problems that is similar to grouping: converting a flat list of nodes to a more hierarchical structure.

Converting Flat to Hierarchical Structure

Many HTML documents, especially those converted from other formats or from database output, have a flat structure with several levels of divisions. The divisions can be indicated by the `h1 . . . h6` header elements or by some other patterns of HTML elements and attributes. It is frequently desirable to convert such documents into a hierarchical XML structure. The task is to wrap all material between level 1 divisions into top-level elements; within those, wrap all material between level 2 divisions into next level elements and so on.

An elegant solution to this problem was posted by Jeni Tennison (`http://jenitennison.com`), elaborating on a suggestion by Wendell Piez (both frequent contributors to xsl-list—see Appendix D for the URL). The idea is to use keys in which each element that signals division at level N is keyed by the result of applying `generate-id()` to the first preceding element that signals a higher-level division:

```
generate-id(("elements that signal divisions at level N-1")[1])
```

For instance, if divisions are signaled by `h1 . . . h6`, one of the keys will be

```
<xsl:key name="next-headings" match="h2"
        use="generate-id(preceding-sibling::h1[1])" />
```

and similarly for `h3..h6`. (Remember that the `preceding-sibling` axis is ordered from the current element backwards, so the argument to `generate-id()` produces the first `h1` element that precedes the current element.) To provide for the possibility that some level divisions may be missing, Tennison's stylesheet actually creates keys in which each element that signals division at level N is keyed by

```
generate-id("elements that signal divisions at all levels higher than N")[1]
```

as in

```
<xsl:key name="next-headings" match="h6"
use="generate-id(preceding-sibling::*[self::h1 or self::h2 or
          self::h3 or self::h4 or self::h5][1])" />
```

The entire stylesheet can be found at `http://jenitennison.com/xslt`, in the Complex Tasks/Constructing hierarchies section. (The site has many other useful examples.) We are going to use a similar technique in a more general case when levels are indicated by arbitrary XSLT patterns. Our material will come from *The World Factbook*, a large public-domain store of information published annually by the Central Intelligence Agency of the U.S. Government.

The CIA World Factbook

Whatever else one might think of the CIA (and we have seen reasonable people disagree on the subject), their annual *World Factbook* is a very useful public service. (See `http://www.cia.gov/cia/publications/factbook/index.html`.) It provides information, in a uniform format, about all the world countries and many non-countries or not-yet-countries (such as the Arctic region, Taiwan, the West Bank, and the Western Sahara, to name a few). Each record is an HTML file with names such as al.html for Albania or nz.html for New Zealand. Each file has links to image files with the country's map and flag , and otherwise is divided into nine categories:

- Introduction
- Geography
- People
- Government
- Economy
- Communications
- Transportation
- Military
- Transnational Issues

The categories are further divided into fields, such as Terrain (within Geography), or Ethnic Groups (within People). A complete listing of fields is provided, in which each field name is a link to a complete listing of the field's values for all countries. For instance, for Ethnic Groups, we get

```
Afghanistan:
Pashtun 38%, Tajik 25%, Uzbek 6%, Hazara 19%,
  minor ethnic groups (Aimaks, Turkmen, Baloch, and others)
```

```
Albania:
Albanian 95%, Greeks 3%, other 2%
  (Vlachs, Gypsies, Serbs, and Bulgarians) (1989 est.)
  note: in 1989, other estimates of the Greek population ranged from 1%
   (official Albanian statistics) to 12% (from a Greek organization)

Algeria:
Arab-Berber 99%, European less than 1%
```

and so on, all the way to Zimbabwe and Taiwan.

Some fields are further divided into subfields. For instance, the field of Land boundaries within Geography is shown as (Bangladesh data):

```
<p><b>Land boundaries:</b>
<br><i>total:</i>
4,246 km
<br><i>border countries:</i>
Burma 193 km, India 4,053 km
```

The entire collection, with maps and flags, can be downloaded as a 64.5MB zip file and used without restrictions. We include their New Zealand file as groupXSLT/nz.html as a starting point for running examples.

The HTML File Structure

An inspection of the HTML source reveals a very regular structure, probably generated from a database. Ignoring the links (external, to the Factbook and the CIA homepages, and internal, to various sections), the divisions are indicated as follows:

- The flag and the map are in a top-level table (`html/body/table`) in the beginning of the document.
- The name of the country is the content of `/html/body/div/h3`, the only such element in the file.
- Category divisions are indicated by `/html/body/center/table/tr/td/b/a[@idand @name]` (that is, by an `<a>` element that has both `id` and `name` attributes). The content of the `<a>` element is the name of the category.

- Fields start at /html/body/p/b. If the field has no subfields, the <b> element is followed by text. The content of the <b> element provides the name of the field, and the following text provides the field content. Although the <p> element containing a field has a fixed structure, there is no way to specify it in a DTD because the element has a mixed-content model. The structure can be specified in a RELAX NG grammar or an XML schema. (See Chapter 8).

- If a field has subfields, the beginning of each subfield is indicated by
<i> . . . </i>., where the content of the <i> element is the name of the subfield. (For an example, see Land boundaries of Bangladesh.)

Given this regular structure, an XSLT can convert flat HTML files (appropriately *Tidy*ed) into hierarchical XML, which can then be queried in a variety of ways beyond those provided by the CIA.

Building a Hierarchy with Keys

For this example, we will keep the XHTML vocabulary intact and simply wrap each category, field, and subfield into a <div> with an appropriate value of the class attribute. The contents of a field or subfield are wrapped into a <span> with the same class attribute. But first we have to define the keys, as here in Listing 6-7 for groupXSLT/ciakey.xsl.

Listing 6-7. Building Hierarchies Using xsl:key *and* generate-id()

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<!-- sections (categories) are children of the country name -->
<xsl:key name="children"
   match="/html/body/div/table/tr/td/b/a[@id and @name]"
                  use="generate-id(/html/body/div/h3)" />
<!-- each subsection (field) is a child of the preceding section (category) -->
<xsl:key name="children"
   match="/html/body/p/b"
   use="generate-id((../preceding-sibling::div/
                        table/tr/td/b/a[@id and @name])[last()])" />
<!-- each subsubsection (subfield)
      is a child of the preceding subsection (field) -->
<xsl:key name="children"
              match="html/body/p/i"
              use= "generate-id(preceding-sibling::b)" />
```

Note that we can have more than one key of the same name (which is what we do here) for greater uniformity of code. We key everything by strings generated by `generate-id()`, and we access the keyed information by the same strings.

The remainder of the groupXSLT/ciakey.xsl stylesheet contains four templates: for country, category, field, and subfield. (See Listing 6-8.) In the country and category templates, we output a `<div>` and within it the name of the country or category. In the field and subfield templates, we additionally output content in a span.

Listing 6-8. The Templates for Building Hierarchies Using `xsl:key` *and* `generate-id()`

```
<xsl:template match="/">
<html><head><title>CIA Structures</title></head><body>
<xsl:variable name="country" select="/html/body/div/h3/text()"/>
<div class="country" name="{$country}">
  <h3><xsl:value-of select="$country"/></h3>
  <xsl:apply-templates mode="section"
     select="key('children',$country/..)"/>
</div>
</body></html>
</xsl:template>

<xsl:template match="a" mode="section">
<div class="category" name="{text()}" style="margin:10">
  <h4><xsl:value-of select="text()"/></h4>
  <xsl:apply-templates mode="subsection"
      select="key('children',.)"/>
</div>
</xsl:template>

<xsl:template match="b" mode="subsection">
<div class="field" name="{text()}" style="margin:10">
  <h5><xsl:value-of select="text()"/></h5>
  <span class="field">
    <xsl:value-of select="following-sibling::text()[1]"/>
  </span>
  <xsl:apply-templates mode="subsubsection"
      select="key('children',.)"/>
</div>
</xsl:template>

<xsl:template match="i" mode="subsubsection">
 <div class="subfield" name="{text()}" style="margin:10">
```

```
    <i><xsl:value-of select="text()"/></i>
   <span class="subfield">
     <xsl:value-of select="following-sibling::text()[1]"/>
   </span>
  </div>
</xsl:template></xsl:stylesheet>
```

Building a Hierarchy Recursively

Listing 6-8 works from the inside out: for each division marker, we store in a key the immediately preceding division marker of the next level up. We ensure that we store and retrieve the same node by keying them with `generate-id()`. To contrast this programming style with the preceding chapter, we will show how the same result can be achieved by forward recursion at the appropriate level of the hierarchy. Whereas earlier examples of recursion used a space-separated list of tokens as the "recursion driver" (see, for example, Listing 5-13), in Listing 6-9 we use node-sets and the `following-sibling` axis for the same purpose. In outline:

- The root template outputs the top-level `div` and the name of the country, then applies templates to all first-level sections (categories).

- The next template processes categories (`match="a" mode=section"`) and within each category calls the recursive `subsections` template to process fields. The parameter is the node-set of all following siblings.

- The subsections template locates each subsection and calls `apply-templates` on each. Because we want our forward processing to stop when the start of the next higher-level sections is found, we cannot simply do `xsl:for-each select="following-sibling::*`.

- The sub-subsections (subfields) template, by contrast, does want to process all the following siblings that are `<i>` elements, and therefore uses `xsl:for-each`. You may call this "opportunistic programming."

The entire stylesheet is shown in Listing 6-9, groupXSLT/ciarecursion.xsl.

Listing 6-9. Building Hierarchies by Forward Processing

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
<html><head><title>CIA Structures</title></head><body>
<xsl:variable name="country" select="/html/body/div/h3/text()"/>
<div class="country" name="{$country}">
```

```
  <h3><xsl:value-of select="$country"/></h3>
  <xsl:apply-templates mode="section"
     select="key('children',generate-id($country/..))"/>
</div>
</body></html>
</xsl:template>

<xsl:template match="a" mode="section">
<div class="section" name="{text()}" style="margin:10">
  <h4><xsl:value-of select="text()"/></h4>
  <xsl:apply-templates mode="subsection"
      select="key('children',generate-id(.))"/>
</div>
</xsl:template>

<xsl:template match="b" mode="subsection">
<div class="subsection" name="{text()}" style="margin:10">
  <h5><xsl:value-of select="text()"/></h5>
  <p>
    <xsl:value-of select="following-sibling::text()[1]"/>
  </p>
  <xsl:apply-templates mode="subsubsection"
      select="key('children',generate-id(.))"/>
</div>
</xsl:template>


<xsl:template match="i" mode="subsubsection">
 <div class="subsubsection" name="{text()}" style="margin:10">
   <i><xsl:value-of select="text()"/></i>
  <p>
    <xsl:value-of select="following-sibling::text()[1]"/>
  </p>
 </div>
</xsl:template>
  </div>
</xsl:template></xsl:stylesheet>
```

With this example, we conclude the subject of efficient grouping and hierarchy building, and return to the issues of recursion.

List Processing and Recursion Depth

As we just mentioned, our earlier examples of recursion used a space-separated list of tokens as the "recursion driver" whereas Listing 6-9 uses node-sets and the `following-sibling` axis for the same purpose. The recursive pattern is the same in both cases: to process a list of items, we first apply a problem-specific predicate to see whether there are more items to process; if the answer is yes, we do what needs to be done with the first item on the list and recursively process the tail end of the list, without the first item.

The common types of list processing are usually grouped into three categories: map, filter, and accumulate. To take a simple example, suppose that we have list of integers from 1 to 10.

- An example of mapping would be a function that takes that list as an argument and returns a list of squares of all the numbers on the list: 1,4,9, . . . 100. This is often described as "mapping the `square()` function over the list."

- An example of filtering would be a function that takes that list as an argument and returns a list of all the prime numbers on the list: 2,3,5,7. This is often described as "filtering the list by the prime() predicate", that is, the predicate that returns True if its argument is a prime number.

- An example of accumulation would be a function that takes that list as an argument and returns the sum of all the numbers on the list. Another example would be finding the maximum element of the list. The definitive feature of accumulation is that the result is not a list but rather a primitive value obtained by performing some computation on all elements of the list.

These three processing patterns can be combined in various ways: we can get the sum of squares of all prime numbers on our list by applying the `prime()` filter, the `square()` map, and the `sum()` accumulator, in that order. The ability to create processing pipelines out of standard functional components is a very powerful feature of list processing.

Another common list-related task is generating a list according to some specification. A simple example is a function that takes two integers, `lo` and `hi`, and generates a range of integers from `lo` to `hi`.

In many languages, lists can be generated and processed using either iteration (a loop) or recursion. In XSLT, some list operations can be done using either `xsl:for-each` or recursion; others can be done only by recursion. In this section, we will show simple examples of list processing in XSLT. In the next section, we will look at alternatives and compare them for time efficiency and the size of the lists that they can generate or process.

List Processing in XSLT

If your list is a node-set and you already have it constructed (usually extracted from data), then `xsl:for-each` is very well suited for filtering and mapping. Consider the `for-each` element of the last template of Listing 6-9:

```
<xsl:for-each select="following-sibling::i">
<div class="subsubsection" name="{text()}">
  <h6><xsl:value-of select="."/></h6>
  <p><xsl:value-of select="following-sibling::text()[1]"/> </p>
</div>
</xsl:for-each>
```

We start with the list of following siblings, filter it by the predicate `name()="i"`, and apply a function to each surviving node that maps it to the appropriate XHTML structure.

If you need to do some sort of accumulation, or if your list needs to be generated in the first place, then `xsl:for-each` is not of much help. We will do a sequence of `range()` examples in the next section, but here is an example of accumulation.

Find Longest String in a List of Strings

Listing 6-10 is a generic template used in max-range/longestverse0.xsl that has a `list` parameter, assumed to be a node-set; it returns a copy of the node that has the longest string value. It follows the familiar recursive pattern. Note how recursion eliminates the need for assignment: instead of changing the value of a variable, we call another instance of the same template with updated values of `$max` and `$maxlen`.

Listing 6-10. Template to Find Node with Longest String Value

```
<xsl:template name="longest">
  <xsl:param name="maxlen" select="0"/>
  <xsl:param name="max" select="''"/>
  <xsl:param name="list" select="''"/><!-- list to process -->
  <xsl:choose>
    <xsl:when test="not($list)"> <!-- list is empty, return copy of $max -->
     <xsl:copy-of select="$max"/>
    </xsl:when>
    <xsl:otherwise>
     <xsl:variable name="first" select="$list[1]"/>
     <xsl:variable name="firstlength" select="string-length($first)"/>
```

```
    <xsl:choose>
      <xsl:when test="$maxlen > $firstlength"><!-- max is unchanged -->
        <xsl:call-template name="longest">
          <xsl:with-param name="max" select="$max"/>
          <xsl:with-param name="maxlen" select="$maxlen"/>
          <xsl:with-param name="list" select="$list[position()!=1]"/>
            <!-- recursively process list without first node -->
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise><!-- new max found -->
          <xsl:call-template name="longest">
            <!-- the new max parameter is $first,
                   converted from result tree to node-set -->
            <xsl:with-param name="max" select="xalan:nodeset($first)"/>
            <xsl:with-param name="maxlen" select="$firstlength"/>
            <xsl:with-param name="list" select="$list[position()!=1]"/>
        </xsl:call-template>
    </xsl:otherwise>
    </xsl:choose>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>
```

Let's use this template to find the longest verse in the book of Genesis. Because we are going to use ot.xml and nt.xml several times in the rest of the chapter, here are a few lines from their DTD, tstmt.dtd:

```
<!ELEMENT tstmt     (coverpg?,titlepg?,preface?,(bookcoll|suracoll)+)>
...
<!ELEMENT bookcoll  (book|sura)+>
<!ELEMENT book      (bktlong, bktshort, epigraph?, bksum?, chapter+)>
...
<!ELEMENT bktlong   (%plaintext;)*>
<!ELEMENT bktshort  (%plaintext;)*>
<!ELEMENT chapter   (chtitle, chstitle?, epigraph?, chsum?, (div+|v+))>
...
<!ELEMENT div       (divtitle, v+)>
```

In summary, a "testament" consists of book collections; book collections consist of books or *suras*; books have long and short titles followed by a number of verses (except for one "chapter," Psalm 119, that consists of `div` elements that contain `v` elements). Given this structure, the highlighted lines in the following

code will call our named template to find the longest verse in the book of Genesis, whose position in Genesis is less than or equal to a given limit:

```
<xsl:param name="limit" select="100"/>
< xsl:template match="/">
<html><head><title>Longest Verse in Genesis</title></head><body>
The longest verse is <p>
<xsl:call-template name="longest">
  <xsl:with-param name="list"
          select="(/tstmt/bookcoll/book[contains(bktlong,'GENESIS')]/chapter/v)
                          [position() &lt;= $limit]"/>
</xsl:call-template>
</p></body></html>
</xsl:template>
```

Can this template be used to find the longest verse in the entire ot.xml file? In a world without friction or memory limitations, the change would be trivial: simply remove the `contains()` predicate in the `select` expression and the `$limit` clause. However, on many installations, the simple recursion pattern will not work because the XSLT processor will run out of memory (specifically, stack memory).

Stack Memory and Recursion Depth

In most implementations of recursive functions (or templates), the program memory is divided in two parts: a heap, for overall storage of objects, and a stack that is used to keep track of scalar values and references to objects. Every time that there is a new recursive call, new values of parameters are placed on the stack and removed only after the call is completed. The maximum number of recursive calls that have to be simultaneously present on the stack is called the *recursion depth*. If recursion is too deep, the program will run out of stack memory (although heap memory may still be available).

As with time efficiency, we estimate the stack memory requirements of a program as a function of the size of input. If a recursive call cannot be completed until the next one is, then the stack will grow in proportion to (as a linear function of) the total number of recursive calls. We call such examples of recursion *linear recursion*. If the template of Listing 6-10 were used to find the longest verse in ot.xml, the total number of recursive calls is the total number of verses in ot.xml, which is too many for most PCs to handle.

Linear Recursion vs. Tail Recursion

In processing the template of Listing 6-10, there is really no reason to keep a recursive call on stack once the next call is made, because the parameters of the next call carry all the information that is needed to complete the computation. In particular, they have the length of the longest-verse element found so far, a pointer to that element, and a pointer to the remaining list to process. A recursive process in which each recursive call carries all the information that is needed to complete the computation is called *tail recursion*. A processor that understands tail recursion can process it very efficiently: for the longest-verse problem, the stack would contain, throughout the computation, only a number (the length of the longest verse so far) and pointers to two objects on the heap—one to the remaining list to process and the other to the longest verse. (More precisely, the second pointer would be to the node-set that contains a single node that holds the longest verse found so far.) GNU C compiler has been doing this kind of optimization for many years, and some XSLT processors (most notably Michael Kay's Saxon) do it, too; Xalan, MSXML, and the rest will follow eventually. There are actually two issues here: one is recognition of tail recursion in itself, which limits stack depth, and the other is recognition of the pattern `$list[position()!=1]` in the recursive call. Saxon can recognize that this amounts to following a list pointer and that it is a job to be done in constant time—but, if the list is stored in an array, forming this value takes linear time. Thus, in Saxon, we might expect to apply this stylesheet in linear time, linear heap space, and constant stack depth; in Xalan, we might expect quadratic time (a linear number of linear operations), linear heap space at any given moment but quadratic total space (a linear number of linear-length lists to be formed and thrown away by the pattern just mentioned), and linear stack depth. For each stylesheet problem, it's worthwhile to think through these four factors: total time, maximum stack depth, maximum heap space, and total heap space. XSLT implementers will make progress, but the implementer has no perfect choice to make. In the meantime, if we want to find the longest verse in the Bible, we have to pursue other options. One is to restructure the stylesheet so as to reduce recursion depth, and the other is to use an extension function (Java, JavaScript, or VBScript) to get the job done. We will investigate both options on the simpler problem of generating a long list before returning to the problem of finding the maximum-length element.

Generating Large Data Sets

In this section, we show how XSLT can be used to generate large XML data sets of a fairly flat regular structure, similar to XML-ized database output. We often need such data sets for testing, but the ways in which they are generated are interesting in their own right. Two useful techniques are shown: breaking a large problem

into smaller chunks (a strategy known as *divide and conquer*) and using external functions to optimize performance. In the examples in this section, we also show how to perform fine-grained time/space measurements using Java method calls.

Generating a Range of Numbers

To make the problem simple and uniform, our data sets are number ranges. Although simple and specific, the task of generating a range of numbers as nodes in an XPath tree actually has many applications because it makes an equivalent of a For loop available in XSLT. If you have a `range` variable whose value is a node-set such that each node's text is (the string representation of) an integer, and those integers range from 1 to 100,000, then you can evaluate an expression 100,000 times by saying

```
<xsl:for-each select="$range">
<!-- expression to evaluate -->
</xsl:for-each>
```

Because of its general usefulness, the `range()` extension function is provided in Michael Kay's Saxon but not in other major XSLT processors. We implement this function in two different ways in XSLT and as an extension function in Java, and compare the performance of the three implementations.

Linear Recursion vs. Tree Recursion (Binary Split)

A straightforward template to output a range of numbers could adhere to the following algorithm.

To output the range of numbers from `lo` to `hi`:

- If `hi < lo`, output nothing and stop.
- Otherwise, output `lo`, then **output the range of numbers from `lo+1` to `hi`.**

The highlighted part of the algorithm is the recursive call. This is another example of linear recursion: the growth of the stack is a linear function of the size of the range. In XSLT, the algorithm (used in `max-range/linearrange.xsl`) comes out as shown in Listing 6-11.

Listing 6-11. Range by Linear Recursion

```
<xsl:template name="range">
  <xsl:param name="low" select="1"/>
  <xsl:param name="high" select="0"/>
  <xsl:if test="$high &gt;= $low">
    <td><xsl:value-of select="$low"/></td>
    <xsl:call-template name="range">
      <xsl:with-param name="low" select="$low + 1"/>
      <xsl:with-param name="high" select="$high"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

A more complicated algorithm would be to follow the strategy of splitting the problem in two equal parts. (A good descriptive name for this strategy is *binary split.*) As before, the highlighted phrases in the last bullet that follows are recursive calls:

To output the range of numbers from `lo` to `hi`:

- If `hi > lo`, output nothing and stop.
- If `hi = lo`, output `lo` and stop.
- Otherwise, find the midpoint between `lo` and `hi`, **output the range of numbers from `lo` to `mid-1`,** output the `mid` number, and **output the range of numbers from `mid+1` to `hi`**.

This algorithm is an example of a general "divide-and-conquer" problem-solving strategy: to solve a problem, you divide it into smaller subproblems, solve those, and combine the results. When implemented recursively (as in Listing 6-12), it results in a pattern called *tree recursion* because each recursive call gives rise to two more calls, and all together the recursive calls form a tree. In Figure 6-3, "R()" stands for a recursive call and "O()" stands for "output a single number."

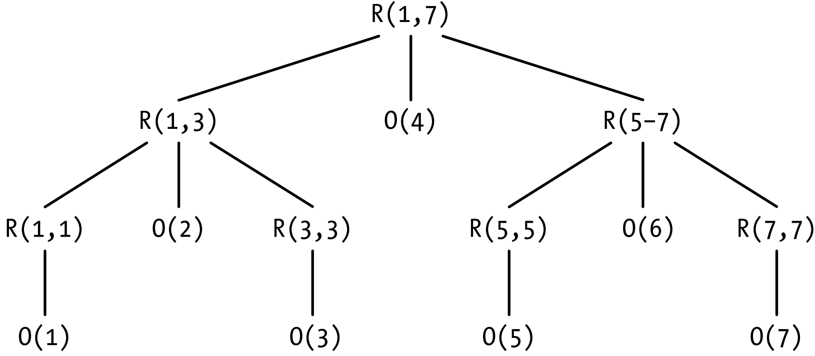

Figure 6-3. A tree of recursive calls for `Range(1-7)`

Listing 6-12 shows max-range/binaryrange.xsl, the tree-recursion version of the range template.

Listing 6-12. Range by Binary Split

```
<xsl:template name="range">
    <xsl:param name="low" select="1"/>
    <xsl:param name="high" select="0"/>
    <xsl:choose>
      <xsl:when  test="$low = $high"><!-- only one number in range; output -->
        <td><xsl:value-of select="$low"/></td>
      </xsl:when>
      <xsl:when  test="$low &gt; $high" /><!-- no numbers in range; do nothing -->
      <xsl:when  test="$low &lt; $high">
         <xsl:variable name="mid" select="floor(($low + $high) div 2)"/>
          <!-- find the mid point -->
         <xsl:call-template name="range"><!-- first recursive call -->
            <xsl:with-param name="low" select="$low"/>
            <xsl:with-param name="high" select="$mid - 1"/>
         </xsl:call-template>
        <td><xsl:value-of select="$mid"/></td><!-- output the middle number -->
         <xsl:call-template name="range"><!-- second recursive call -->
            <xsl:with-param name="low" select="$mid + 1"/>
            <xsl:with-param name="high" select="$high"/>
         </xsl:call-template>
     </xsl:when>
    </xsl:choose>
  </xsl:template>
```

The advantage of tree recursion over linear recursion is that its recursion depth is in proportion to the logarithm of the size of the problem. Even though the total number of recursive calls is the same as in the linear version, at any given time the stack contains at most the logarithm of the range size, because recursive calls in the tree are processed in the depth-first order. Here is the order of execution for R(1,7):

```
R(1,7)
R(1,3) O(4) R(5-7)
R(1,1) O(2) R(3,3) O4 R(5-7)
O(1) O(2) R(3,3) O(4) R(5-7)
O(1) O(2) O(3) O(4) R(5-7)
...
```

The maximum number of R's on the stack is 3, the logarithm of 8. For R(1,1000), the maximum stack size would be 10, as opposed to 1,000 for the linear version. Logarithms grow much slower than a linear function does, so a tree recursive procedure should be able to generate much larger ranges than a linear-recursive one. Is this indeed true? The next section shows a framework for testing whether all this fine theory is confirmed in practice.

Comparison

To test, we are going to run both algorithms on ranges of different size, each time asking these questions:

- Is the processor up to it? Obviously, as the range is increased, at some point the answer will be "no": the program will either generate an error message, die silently, or crash the computer. Will that point be the same for linear recursion and tree recursion approaches?

- How long does it take to do it? If possible, we would like to measure how much time passes specifically from the moment that the `range` template is called the first time to the moment it returns.

To take the scoop out of it, Table 6-1 shows the results for the binary-split and linear-recursion versions, as run in Xalan. The smaller numbers are basically noise: it doesn't reliably take less time to generate a three-item list than it does for a one-item, and it doesn't reliably take more time to generate 100 than 300 by linear recursion, but overhead depends on the state of the XSLT engine and of other processes in the machine. As you can see, the linear recursion version stops at 3,000 because it just couldn't do the 10,000 range in our testing configuration,

whereas binary-split did it quite easily, and in fact went on to produce the 1–100,000 range. The absolute values will of course vary from one machine to another, but the relative time/space efficiency should remain about the same.

Table 6-1. Time and Space for Range Generation in XSLT

	BINARY SPLIT		LINEAR RECURSION	
LIMIT	TIME (MSEC)	MEMORY (BYTES)	TIME (MSEC)	MEMORY (BYTES)
1	270	22728	280	16976
3	220	11472	170	11472
10	220	24824	220	23152
30	280	39672	280	51848
100	330	85504	390	108400
300	220	217360	220	340912
1000	380	657648	330	1483920
3000	600	2261728	720	7318400
10000	1540	8213448		
30000	4340	38870016		
100000	24000	42921800		

Measurement Code

Our measurements for Table 6-1 have been made in an all-Java environment: both the XSLT processor and the extension functions that measure time and space usage are written in Java. This means that they run in the same process, the Java virtual machine. This is good for comparing the relative merits of different algorithms. However, as we indicated, different processors can be optimized in different ways; in particular, linear recursion can be made to use space more efficiently.

We measure memory use in addition to timing. The figures we obtain are for total memory (heap and stack together) because Java does not provide tools to measure stack memory specifically. However, the code may be useful. Its output shows that the space requirements for small ranges look rather random: most of the space consumption is actually overhead. For larger ranges, space consumption is roughly proportional to the size of the range, as might be expected.

Note that these figures show total memory use and that they are rough approximations.

The complete XSLT stylesheets that run these range templates on multiple inputs and measure performance are max-range/range.xsl and max-range/rangebin.xsl, Listing 6-13 through 6-15. Because you may well want to edit these to run to the limits of your particular implementation, we suggest that you run them outside of the webapp environment; for Windows environments you'll find batch files with matching names (such as max-range/range.bat.) Simply double-click the batch file to generate the corresponding output file (such as max-range/rangeout.xsl.) The listings omit explanatory text with HTML markup.

Listing 6-13. Range Measurements 1: Namespaces

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
   xmlns:java="http://xml.apache.org/xslt/java"
   xmlns:xalan="http://xml.apache.org/xalan"
   xmlns:dat="http://n-topus.com/ns/data"
   exclude-result-prefixes="xalan java dat">
<xsl:output method="html"/>
```

We declare three namespaces, in addition to the XSLT namespace. We refer to them by their prefixes for convenience. The `java` namespace is for extension functions written in Java, as explained in the next section. All the functions in this example come from our XslUtil class. The `xalan` namespace is for the Xalan-specific extension function that converts result-tree fragments to node-sets. The `dat` namespace is for XML data that we are going to include in the stylesheet itself. (Remember that an XSLT stylesheet is an XML document that can contain any XML data.) The stylesheet will extract data out of the stylesheet using the `document()` function with the empty string as argument. (See Listing 6-14.)

Listing 6-14. Range Measurements 2: Limits and the `document()` *Function*

```
<dat:limits>
<!-- limits for linear recursion shown; binary split has more of them -->
   <d>1</d><d>3</d><d>10</d> <d>30</d><d>100</d>
   <d>300</d><d>1000</d> <d>3000</d>
</dat:limits>
<xsl:template name="range">
  <!-- The range template of Listing 6-11 or 6-12 goes here -->
</xsl:template>
<xsl:template match="/" >
<html><head><title>tst</title>XSLT Measurements</head><body>
<!-- output the top row of table -->
<table border="1" >
<tr><th>Limit</th><th>Time(in msec)</th><th>Space</th><th>Count</th></tr>
<!-- extract limits data from the stylesheet -->
<xsl:variable name="limitlist" select="document('')/xsl:stylesheet/dat:limits/*">
```

The rest of the stylesheet outputs the table. For each item on the `limitlist`, we perform the same actions:

- clear memory
- register start time and the amount of memory in use
- run the range template
- register end time and the amount of memory in use
- convert the generated range from result-tree fragment to node-set and count it

The value of the current limit, time and space measurements, and the count fill in the current row of the table, as shown in Listing 6-15.

Listing 6-15. Range Measurements 3: Table Output

```
<xsl:for-each select="$limitlist*"><!-- for each limit -->
  <xsl:variable name="limit" select="."/>
  <xsl:variable name="dummy-void" select="java:XslUtil.clearMemory()"/>
  <xsl:variable name="startTime" select="java:XslUtil.time()"/>
  <xsl:variable name="startMemory" select="java:XslUtil.usedMemory()"/>
  <xsl:variable name="theRange">
    <!-- construct the current range as value of variable -->
    <xsl:call-template name="range">
      <xsl:with-param name="high" select="$limit"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="endTime" select="java:XslUtil.time()"/>
  <xsl:variable name="endMemory" select="java:XslUtil.usedMemory()"/>
  <tr><td align="right"><xsl:value-of select="$limit"/></td>
    <td align="right"><xsl:value-of select="$endTime - $startTime"/></td>
    <td align="right"><xsl:value-of select="$endMemory - $startMemory"/></td>
    <td align="right"><xsl:value-of
                     select="count(xalan:nodeset($theRange)/*)"/></td>
  </tr>
</xsl:for-each>
</table></body></html>
</xsl:template></xsl:stylesheet>
```

As you can see, extension functions are a powerful tool. We will explain how they work after presenting the specific Java functions used in this example.

Java System Functions for Time and Memory

The Java-XSLT utilities of this section are one-liners that call Java system functions. We have collected them into a class of our own for convenience and clarity. The file is xmlp/WEB-INF/classes/XslUtil.java. If at some point you want to change and reuse it, you'll find a make.bat file in the same directory: stop the Web server, double-click on make.bat, and you'll see that it compiles the file, puts it into the standard "jar" distribution format, and copies it into Tomcat's common/lib directory. (If you're not using Tomcat on Windows, you'll have to edit the make file to do what you want.) Restart the Web server and run your code. The details of Java code are completely immaterial for our topic (XSLT efficiency), but, for those who are familiar with the language, we should mention that it is easiest to run a Java function from XSLT if it is a public static method of a public class, which is why all the methods of XslUtils are declared public and static. (See Listing 6-16.)

Listing 6-16. System Functions in XslUtil

```
public static void clearMemory(){// call the Garbage Collector
  System.gc();
}
public static double freeMemory(){ // return the amount of free memory
  return (double)Runtime.getRuntime().freeMemory();
}
public static double usedMemory(){// return the amount of memory in use
  Runtime runtime = Runtime.getRuntime();
  return (double)(runtime.totalMemory() - runtime.freeMemory());
}
public static double time(){ // return current time
   return System.currentTimeMillis();
}
```

Summary of Results

To summarize the findings of this section, binary split and linear recursion take similar time (both generate the same number of recursive calls, after all), but binary split uses much less stack memory and can produce much larger data sets. We have also experimented with a ten-way split algorithm that recursively splits the problem into ten subproblems (rather than two, as in binary split). Somewhat to our surprise, its performance was no better than that of the binary split, and the code (max-range/range10.xsl, included with the book's code) is considerably more complicated.

Extension Functions

Extension functions are functions written in a language other than XSLT and called from within XSLT. The most common languages to use are Java, JavaScript, and various scripting languages, including VBScript.

XSLT 1.0 provides no specification for how a function call in XSLT is bound to the appropriate binary object outside it. However, vendor-specific implementations are fairly consistent with each other, and *XSLT 2.0* will definitely build on them to provide a standard set of conventions as well as bindings for specific languages. (Java and JavaScript are the prime candidates.) The common theme is that a binding for external functions is associated with a namespace, and each function call is an XML element in that namespace. You have seen (in Listing 6-15) such examples as

```
<xsl:value-of select="count(xalan:nodeset($theRange)/*)"/></td>
<xsl:variable name="endMemory" select="java:XslUtil.freeMemory()"/>
```

Currently, the namespace URIs for extension functions are XSLT processor specific, but, if your language is Java and you map the namespace to the `java:` prefix, then simply replacing the namespace will switch your code from Xalan to Saxon or the Oracle XSLT processor. Of course, if your functions use processor-specific class libraries (as some of ours do), then more conversion will be needed.

When Are Extension Functions Useful?

The most common uses of extension functions are to get access to system services, optimize performance, and get access to an external non-XML data source, such as a database.

We have seen the first of these in the preceding section. The second one is shown in this section, and the third will be shown in the next chapter.

Generating a Range Using an Extension Function

A natural question to ask is whether the performance will improve if a large node-set, such as the number ranges of the preceding section, is generated by an external function. We have experimented with external functions in Java, JavaScript, and VBScript.

Range Function in Java

To experiment with a Java extension function, all you need to do is change the way in which the range is generated in Listing 6-15. Instead of

```
  <xsl:variable name="theRange">
  <!-- construct the current range as value of variable -->
    <xsl:call-template name="range">
      <xsl:with-param name="high" select="$limit"/>
    </xsl:call-template>
</xsl:variable>
```

we now have (in rangejava.xsl)

```
  <xsl:variable name="theRange" select="java:XslUtil.range(1,$limit)"/>
```

The `range()` method of XslUtil actually returns an object that implements the DOM NodeList interface. The conversion of DOM NodeList to XPath node-set is done automatically by Xalan.

To construct a NodeList object, you construct a sequence of children of a dummy element; in the end, you return that sequence using the `getChildrenNodes()` method. In our case, each child node is of type `<item>` and itself has a child node of type `text` that contains the integer value converted to string. See Listing 6-17.

Listing 6-17. System Functions in XslUtil, Used in Listing 6-14 and 6-15

```
import org.apache.xpath.axes.*;
import org.apache.xpath.*;
import org.w3c.dom.*;
public class XslUtil {
// a number of XSLT utilities,
// including range(), freeMemory(), clearMemory() and time()
public static Document  newDoc() throws Exception {
  DocumentBuilderFactory dbf=DocumentBuilderFactory.newInstance();
  dbf.setNamespaceAware(true);
  dbf.setValidating(false);
  return dbf.newDocumentBuilder().newDocument();
}
public static NodeList range(double lo,double hi){
try{
  int ilo=(int)lo;
  int ihi=(int)hi;
  Document doc= newDoc();
```

```
    Element result = doc.createElement("result");
    int ilo=(int)lo; int ihi=(int)hi;
    for(int i=ilo;i<=ihi;i++){
      Node nd=doc.createElement("item");
      result.appendChild(nd);
      nd.appendChild(doc.createTextNode(Integer.toString(i)));
      }
    return result.getChildNodes();
  }catch(Exception ex){ex.printStackTrace(); return null;}
  }
...
} // end of XslUtil class
```

To report the results, we modify Table 6-1 by removing the memory columns and adding new measurements to produce Table 6-2.

Table 6-2. Time for Range Generation by an Extension Function

LIMIT	BINARY SPLIT TIME(MSEC)	LINEAR RECURSION TIME(MSEC)	JAVA EXTENSION FUNCTION TIME(MSEC)
1	270	280	390
3	220	170	160
10	220	220	160
30	280	280	160
100	330	390	160
300	220	220	160
1000	380	330	220
3000	600	720	220
10000	1540		280
30000	4340		490
100000	24000		

In this table, the extension function is a good deal faster than either of the pure XSLT solutions, but it may run out of memory (total memory, not stack) because it uses memory less intelligently than the XSLT processor does. Is this a safe conclusion? Emphatically not: it depends on the details of your XSLT processor implementation, and these will change with every upgrade. In fact, this table has changed drastically as we've gone through the book, upgrading our JDK

as well as Tomcat, Xalan, and Xerces: Speed generally rises, but not consistently, and it is perfectly possible for the binary recursion to win in speed as well as overall size.

Extension Functions vs. External (Web Application) Functions

In addition to extension functions, external code written in other languages can be invoked from within an XSLT stylesheet using the `document()` function. You have already seen such usage in some of our earlier examples. Recall that the `document()` function takes one argument, a URL, and loads the document specified by the URL into the XSLT processor. It is perfectly legal to attach a query string to the URL in the usual way and invoke an application on the server. We illustrate this usage in rangeasp.xsl that builds a range by invoking a VBScript function in an ASP page on an IIS server. Its only difference from the Java version is that instead of

```
  <xsl:variable name="theRange" select="java:XslUtil.range(1,$limit)"/>
```

we now say

```
<xsl:variable
  name="theRange"
  select="document(concat('http://localhost/xml/range.asp?',
                          'lo=1&amp;hi=',
                          $limit))/*"/>
```

The argument of the `document()` function is the following URL (URL-encoded and with the expression `$limit` evaluated):

```
http://localhost/xmlp/range.asp?lo=1&hi=$limit
```

The function returns a complete document tree including a document root. We take its children, of which there is just one, the root of the element tree created by range.asp:

```
<%@ LANGUAGE="VBSCRIPT"
 %><% Option Explicit
 %>
  <list>
<%
  Dim lo,hi
```

```
  lo = Int(Request.QueryString("lo"))
  hi = Int(Request.QueryString("hi"))
  Do While NOT lo > hi
    Response.Write "<d>"
    Response.Write lo
    lo = lo + 1
    Response.Write "</d>"
  Loop
%>
</list>
```

Note that our call on `document()` that invokes an ASP page with VBScript code coexists quite peacefully with all of the Java system function calls that measure performance and memory use. However, the results of those calls may be hard to interpret, because the entire range creation is done out of (Java) process.

Extension Functions vs. External Web Applications

Extension functions are a more general mechanism. When they are standardized, they will be even more useful. The advantages of external functions invoked via `document()` are that their invocation rules are already stable within any given invocation tool (ASP, JSP, and so on), and that their code can reside anywhere on the Internet.

Although limited in scope, external functions are a useful tool for calling non-XSLT code from within XSLT.

The Longest Verse, Revisited

Now that we have new tools, let's revisit the problem that we could not solve using linear recursion: finding the longest verse in the Bible. We present a pure-XSLT solution using tree recursion and three extension-function solutions using Java, JavaScript, and VBScript.

Tree Recursion Based on the Document Tree

Instead of dividing the data in half, our tree-recursion solution (max-range/lvdivconq.xsl) uses the tree structure of the data itself to break the problem into manageable chunks. Within each chapter, we find the longest verse by linear recursion, exactly as in Listing 6-10. Each chapter reports its longest element to the next level of the document tree (book), and the longest element among books is the longest element of the entire Bible.

Recursion via xsl:apply-templates *Within a Variable*

The recursion is performed by three templates that have the same mode attribute, whose value is lv, for "longest verse". (See Listing 6-18. If needed, review the Chapter 5 section "Using Push with the mode Attribute".) The first template is the most general: it matches any element node. It calls xsl:apply-templates within a variable, so the results of template application to the children of the matched node are collected into that variable. The variable value is then sent to the longest template, to identify the winner within the current level of processing.

The other two templates of the lv mode are more specific. One matches text nodes and outputs nothing to prevent default matching and output. The other serves as the base case of the recursion: it matches chapter nodes and outputs the longest verse of each chapter, without invoking xsl:apply-templates. The operation of the program depends on the XSLT conflict-resolution algorithm that, in case of conflict, selects the more specific template.

Listing 6-18. Recursion down the Document Tree

```
<xsl:template match="*" mode="lv">
<xsl:variable name="nextLevel">
  <xsl:apply-templates mode="lv"/><!-- This is the essential recursion. -->
</xsl:variable>
<xsl:call-template name="longest"><
  <xsl:with-param name="list" select="xalan:nodeset($nextLevel)/v"/>
<!--
nodeset($nextLevel) returns a node-set that consists of a single dummy node
whose children are the v (verse) nodes that we want;
nodeset($nextLevel)/v produces the node-set of those verse nodes
 -->
</xsl:call-template>
</xsl:template>
<xsl:template match="chapter" mode="lv">
  <xsl:variable name="verses" select="v"/><!-- no more 'apply-templates' -->
  <xsl:call-template name="longest">
    <xsl:with-param name="list" select="$verses"/>
  </xsl:call-template>
</xsl:template>
<!-- suppress output from default templates -->
<xsl:template match="text()" mode="lv"/>
```

Listing 6-18 assumes that all v nodes are children of chapter nodes. In fact, ot.xml has one "chapter" (Psalm 119) that consists of div elements that contain v elements. We leave it as an exercise to incorporate this detail into Listing 6-18 and the remaining programs of the chapter.

Longest Verse by Tree Recursion

The entire program is shown in Listing 6-19. Its root template starts the process by invoking `xsl:apply-templates` with mode lv.

Listing 6-19. The Entire lvdivconq.xsl Program

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
   xmlns:xalan="http://xml.apache.org/xalan"
   exclude-result-prefixes="xalan">
<xsl:template match="/">
<html><head><title>Longest Bible Verse</title></head><body>
The longest verse is <p>
  <xsl:apply-templates mode="lv"/>
<!-- apply to children of root, of which there is one -->
</p></body></html>
</xsl:template>

<xsl:template match="*" mode="lv">
  <!-- See Listing 6-18 -->
</xsl:template>

<xsl:template match="chapter" mode="lv">
  <!-- See Listing 6-18 -->
</xsl:template>

<xsl:template match="text()" mode="lv"/>  <!-- as in Listing 6-18 -->

<xsl:template name="longest"><!-- identical to Listing 6-10 -->
</xsl:template>
</xsl:stylesheet>
```

This program successfully finds the longest verse. The idea of “recursive descent” into the document tree is applicable to any problem that looks for a summary value computed from a regular XML structure.

Longest Verse Using Extension Elements and Functions

In this section, we present several longest-verse programs that use external functions. Each solution in this section is characterized not just by the language in which it is written but also by the software configuration in which it can run.

Because the programs run as Web applications, the configuration includes a specific server. We have

- lvvbs.xsl, a stylesheet that calls a VBScript function within an ASP application that uses MSXML 3
- lvjs.xsl, a stylesheet that calls a JavaScript function within a JSP application that uses Xalan
- several stylesheets that call a Java function within a JSP application on a Tomcat server that uses Xalan

The reason that we have more than one Java example is that Java provides more options: whereas the JavaScript and VBScript functions can return only a scalar value (a string or a number), Java functions can in addition return a reference to the actual node in the tree, so you can immediately use the value returned by the function in an XPath expression.

Another difference between Java solutions and solutions written in scripting languages is that scripting language code can be placed directly within the stylesheet, as the content of an extension element. An extension element's name belongs to a namespace that has a special meaning to the processor; frequently, extension elements contain executable content. So, for instance, an XSLT processor from XSLTSmiths company (`www.xslt-smiths.com`) can have the following feature: if an XSLT program maps a certain prefix to the `urn:xslt-smith-com-js` namespace URI, the processor will assume that all elements with that prefix contain JavaScript code and process it accordingly. The JavaScript code will, of course, have access to all elements both within the XML data and the XSLT itself. You will see an example of an extension element in a VBScript program shortly.

Just as with extension functions, extension elements will be standardized in *XSLT 2.0*.

VBScript Extension Function with MSXML 3

A VBScript function to locate the longest verse is defined, in a straightforward way, within the `msxsl:script` extension element. In this case, the file is maxrange/longestversevbs.xsl. The element's attributes declare the language in which the code is written and the prefix that will signal the extension function. (See Listing 6-20.)

Listing 6-20. VBScript Extension Function Defined in `msxml:script`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes="ms msxsl">

<msxsl:script language="VBScript" implements-prefix="ms" >
Function maxVerseNumber(vList)
  Dim i,listlen,longestlen,longestloc, item
  listlen = vList.length
  longestlen = 0
  longestloc = -1
  For i = 0 To listlen - 1
    itemStr = vList.item(i).text
    itemlen = len(itemStr)
    If(itemlen &gt; longestlen) Then
      longestlen = itemlen
      longestloc = 1 + i
    End If
  Next
  maxVerseNumber = longestloc
End Function
</msxsl:script>
```

Once the verse is found within the node-set of all verses, we can find its book, chapter, and verse number within the chapter using appropriate path expressions, as shown in Listing 6-21.

Listing 6-21. The Root Template Calling a VBScript Extension Function

```
<xsl:template match="/">
<html><head><title>Longest Bible Verse</title></head><body>
<xsl:variable name="v"   select="/tstmt/bookcoll/book/chapter/v"/>
<xsl:variable name="lvn" select="ms:maxVerseNumber($v)"/>
<xsl:variable name="lvv" select="$v[$lvn]"/>

The Longest verse is number <xsl:value-of select="$lvn"/>, of
length <xsl:value-of select="string-length($lvv)"/>,
which is verse <xsl:value-of select="1+count($lvv/preceding-sibling::*)"/>
in chapter <xsl:value-of select="$lvv/../chtitle"/>
of the book of <xsl:value-of select="$lvv/../../bktshort"/>.
</body></html>
</xsl:template></xsl:stylesheet>
```

JavaScript Extension Element with Xalan

In the case of JavaScript code run by Xalan, there is an additional level of software, Bean Scripting Framework (BSF), that is used to run scripting code within Java applications (such as Xalan). Otherwise, the structure of the program is quite similar to the VBScript solution. The root template is identical except for the highlighted line that invokes the external function, which now reads

```
<xsl:variable name="lvn"  select="jstraverse:maxVerseNumber($v)"/>
```

The function itself is defined within an extension element that is within another extension element: the outer element says “This is an extension component,” and the inner element says “This will be a JavaScript function.” They are all packaged into separate namespaces that are declared on the root element. (See Listing 6-22.)

Another feature of the JavaScript solution is that the NodeList provided to the function is a DOM NodeList object, with the `getLength()` and `item()` methods.

Listing 6-22. JavaScript Extension Function in Xalan

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns:lxslt="http://xml.apache.org/xslt"
                xmlns:jstraverse="JavascriptTraversal"
                extension-element-prefixes="jstraverse"
                xmlns:xalan="http://xml.apache.org/xalan"
                 exclude-result-prefixes="xalan jstraverse lxslt">

  <lxslt:component prefix="jstraverse" elements="" functions="maxVerseNumber">
    <lxslt:script lang="javascript">
      function maxVerseNumber (vList) {
        var listlen= vList.getLength();
        var longestlen=0;
        var longestloc=-1;
        for(var i=0;i &lt; listlen;i++){
          var item= vList.item(i);
          var itemStr= ""+item.getFirstChild().getNodeValue();
          var itemlen=itemStr.length;
          if(itemlen &gt; longestlen){
            longestlen=itemlen; longestloc=1+i;
            }
        }
        return longestloc;
     }
    </lxslt:script>
  </lxslt:component>
```

In summary, to define and use a JavaScript extension function, you have to define the appropriate namespaces (as shown in the highlighted lines of Listing 6-22) and write your JavaScript code inside a `script` element that is, in turn, contained within a `component` element, both in the `http://xml.apache.org/xslt` namespace. For more details on how non-Java extension functions are defined and used with Xalan, see `http://xml.apache.org/xalan-j/extensions.html`. This file is also part of the Xalan distribution, within the `doc` subdirectory of the root. Also in the distribution are a number of samples, including samples of extension functions. To run our max-range/longestverseJS.xsl, you must include the bsf.jar file as scripting-language framework and the js.jar specific JavaScript language; to use other scripting languages, you would use their implementation jars for the BSF.

Because Xalan is itself a Java program, using Java extension functions is easier than using those written in other languages. You have seen Java extension functions in efficiency-measurement examples earlier in the chapter, and the next section provides more examples.

Java Extension Functions

In this section, we present three Java extension functions. All three require the same namespaces to be declared, and the corresponding prefixes excluded from output:

```
xmlns:java="http://xml.apache.org/xslt/java"
xmlns:xalan="http://xml.apache.org/xalan"
exclude-result-prefixes="xalan java"
```

All three functions are used in the same root template that you saw in Listing 6-21. They differ in how they are invoked and what they return. They also differ in the extent to which they depend on Xalan-specific APIs. In the remainder of this section, we show the function call and the code for each of the three functions.

longestLoc()

The first function is an exact analog of the scripting functions of the preceding sections: it takes a NodeList and returns the position of the maximal length node in it.

```
<xsl:variable name="v" select="/tstmt/bookcoll/book/chapter/v"/>
<xsl:variable name="lvn"  select="java:XslUtil.longestLoc($v)"/>
```

The code of this function uses only standard DOM interfaces, as indicated by comments. (See Listing 6-23.)

Listing 6-23. Java Extension Function, Returns a Scalar, Pure DOM

```
public static Double longestLoc (NodeList list) {
 try{
  int listlen=list.getLength(); // DOM
  int longestlen=0;
  int longestloc=-1;
  for(int i=0;i < listlen;i++){
    Node item=list.item(i); // DOM
    String itemStr=item.getFirstChild().getNodeValue(); // DOM
    int itemlen=itemStr.length();
    if(itemlen > longestlen){
      longestlen=itemlen; longestloc=1+i;
      }
  }
  return new Double(longestloc);
 }catch(Exception ex){ex.printStackTrace();return new Double(1);}
}
```

The communication between the XSLT program and this Java function takes place in the following two lines of code from max-range/longestversejava.xsl. The first line constructs the variable to send as an argument to the Java function. The second line returns the result computed by the Java function to XSLT and makes it the value of an XSLT variable.

```
<xsl:variable name="v" select="/tstmt/bookcoll/book/chapter/v"/>
<xsl:variable name="lvn"  select="java:XslUtil.longestLoc($v)"/>
```

maxValLoc()

The next function also returns a double, but it takes two arguments: a node-set and an XPath expression to be applied to each node of the node-set. We return the position of the node that has the maximum value of that expression. To find the position of the longest verse, we use the appropriate expression:

```
<xsl:variable name="v" select="/tstmt/bookcoll/book/chapter/v"/>
<xsl:variable name="lvn" select="java:XslUtil.maxValLoc($v,'string-length(.)')"/>
```

This function is more flexible than the preceding one; for instance, to find the shortest verse, we would simply change the second argument of the function call:

```
<xsl:variable name="lvn"
             select="java:XslUtil.maxValLoc($v,'0 - string-length(.)')"/>
```

However, this flexibility comes at a price: the function uses Xalan-specific XPath APIs, and these APIs are changing. In Listing 6-24, you see that the `XPathAPI.eval` static method is used to apply the XPath string to each successive node, and we take the numeric value of the “eval” result. The only unintuitive part of the method is the highlighted line labeled “Xalan 2.2.0 workaround”, which actually uses almost all of the time spent in this evaluation. Xalan’s internal representation of nodes has changed (for performance reasons) in ways that have left the XPathAPI behind. Our workaround is to create a document (with the “newDoc” method from Listing 6-17) and import each node (really a node “proxy”) into that new document—in effect making a copy of it to which we apply the XPath string. This works, but it’s much slower than it should be; we expect that, by the time you read this book, that line will be unnecessary.

Listing 6-24. Java Extension Function, Returns a Scalar, Uses `org.apache.xpath`

```
public static Double maxValLoc (NodeList list,String xPath) {
 try{
  int listlen=list.getLength(); // DOM
  double maxVal=Double.NEGATIVE_INFINITY;
  int maxLoc=-1;
  Document doc = newDoc(); // see Listing 6-17
  for(int i=0;i < listlen;i++){
    Node item=list.item(i); // DOM
    try{
      item = doc.importNode(item,true); //  Xalan 2.2.0 workaround.
      double itemVal=XPathAPI.eval(item,xPath).num(); // org.apache.xpath
      if(itemVal > maxVal){
        maxVal=itemVal; maxLoc=1+i;
      }
    }catch(Exception ex){}
  }
  return new Double(maxLoc);
 }catch(Exception ex){ex.printStackTrace();return new Double(1);}
}
```

The remaining `max()` function (presented in the next section) returns a node rather than a scalar value. It is thus even more flexible: the returned value can be

plugged directly into an XPath expression. It is also heavily dependent on a specific XSLT processor and a specific programming language. If your application has to work on different processors, then your extension functions should return only scalar values (strings, numbers, and Booleans). On the other hand, something like our `max()` function, similar to Michael Kay's `max()` built into Saxon, will probably become part of the standard extension function library.

max()

This function takes the same parameters as `maxValLoc()`, a node-set and an XPath expression:

```
<xsl:variable name="v" select="/tstmt/bookcoll/book/chapter/v"/>
<xsl:variable name="lvv" select="java:XslUtil.max($v,'string-length(.)')"/>
```

We use the `max()` function to illustrate one more Xalan-specific feature: a node-set can be passed as a parameter to a Xalan extension function as an (unquoted) XPath expression. In other words, `max()` can be called from XSLT code without declaring an auxiliary variable that holds the node-set:

```
<xsl:variable name="lvv"
  select="java:XslUtil.max(/tstmt/bookcoll/book/chapter/v,'string-length(.)')"/>
```

In either case, the returned value is the longest verse itself, not its position in the node-set, and we no longer have to say

```
<xsl:variable name="lvv" select="$v[$lvn]"/>
```

With `max()`, we obtain `lvv` directly. Otherwise, the code (still within `XslUtil.java` and shown in Listing 6-25) is very similar to Listing 6-24.

Listing 6-25. `max()` *Returns a Node*

```
public static Node max (NodeList list,String xPath) {
  Node result=null;
 try{
  Document doc=newDoc(); // Listing 6-17
  int listlen=list.getLength();
  double maxVal=Double.NEGATIVE_INFINITY;
for(int i=0;i < listlen;i++){
    Node item=list.item(i);
    try{
      item = doc.importNode(item,true); // Xalan 2.2.0 workaround
```

```
      double itemVal=XPathAPI.eval(item,xPath).num();
      if(itemVal > maxVal){
        maxVal=itemVal; result=item;
      }
    }catch(Exception ex){}
  }
  return result;
 }catch(Exception ex){ex.printStackTrace();return result;}
}
```

This concludes our discussion of extension functions, and the entire Chapter 6.

Conclusion

The efficient use of resources, especially processing time and stack memory, is an important consideration in XSLT programming. The techniques of this chapter can sometimes make the difference between being able or unable to process a large set of data within specific time and space constraints. However, XSLT is simply a wrong tool for many processing tasks, and these include tasks that are too big, that require extensive processing of document text content, or that produce a lot of new structure on output (rather than rearranging existing structures of the input document). On such tasks, programmatic API for working with XML data in general-purpose programming languages may be a better tool. There are two sets of such APIs—SAX and DOM—and they are the subject of the next chapter.

CHAPTER 7

XML Repository

THIS CHAPTER IS ORGANIZED around a larger application that brings together several topics from earlier chapters and introduces some new ones. The application is a repository of educational materials for a commonly taught college course, to be developed and used cooperatively by educators around the world. The XML data in the repository is actually metadata: citations of books and XML documents, and annotations of those citations and other annotations.

At the heart of the application is what we will call an *XML database*: a small set of potentially large collections of small, similarly structured XML documents. An XML database is analogous to a relational database, which is a small set of tables, each of which is a potentially large collection of small, similarly structured records. We are replacing the traditional flat database row with an XML document, which has a more flexible hierarchical structure. Its greater flexibility creates disadvantages that we're learning to cope with and advantages that we're learning to use. In particular, XPath can be used as a query language on XML records.

The three common ways to implement an XML database are as follows:

- *Store XML data in the file system as a single XML document or several documents (one per "table") or a great number of documents (one per record).* We have done an implementation in which the entire database is persisted as a single XML file; when parsed, each "table" is stored as a vector of DOM trees, each tree corresponding to a record. The code is available with the book's distribution.

- *Store XML data in a relational database.* This is the approach that we will present in detail in this chapter.

- *Store XML in a native XML database.* A native XML database stores namespaces, elements, attributes, and document order in internal data structures optimized for XML data retrieval.

We refer to these three approaches as *vdb*, *rdb*, and *xdb*, respectively.

The first approach, vdb, is the simplest, and it works well for small repositories. What counts as "small" depends, of course, on the amount of available

memory, real or virtual: figure that the DOM trees will require several times as many megabytes as the file itself.

The second approach, rdb, incurs a performance penalty for data transformations between relational and XML formats, but it uses mature technology that supports complex queries and database operations. Within this approach, we will learn how to store XML data in a relational database, how to access a relational database from within an application, and how to convert query results to XML.

The third approach, xdb, may be the best long-term solution, but the technology is still quite new and changing fast, and so we didn't feel it was appropriate to include it in the book at this stage.

In terms of new material, the chapter covers these topics:

- XML languages for representation of metadata: RDF and Dublin Core
- organization of annotations using XLink
- storing XML data in a relational database, combining SQL and XPath queries

The last item is properly the subject of the new standard, *XQuery 1.0: An XML Query Language,* currently being developed by W3C (`www.w3c.org/TR/xquery/`). However, the techniques we present are valuable in other contexts as well.

In outline, this chapter will proceed as follows:

- the structure of XML data, RDF, and Dublin Core
- the structure of the database
- the overall structure of the application
- general issues in database access and specifics of JDBC
- query implementations and XPath filtering

The Structure of XML Data

XML data in our repository is of four kinds: citations, annotations, submissions, and subject keywords. Citations are references to materials that are used in a course, both text and multimedia, hard copy and online. In this simplified version of the project, we work only with text sources and we assume that they are either entire books or online XML documents; we ignore journal articles,

individual chapters in edited volumes, and a host of other variations that would take us too far into library science.

Annotations, broadly understood, include the chapter or page numbers to specify a reading assignment, the topics covered in such an assignment, the approximate class time required to work through an assignment, the description of a classroom activity, essay and exam questions, and so on. Annotations can be linked to sources, other annotations, or both.

Submissions data contains the identity of the submitter and the date of submission. It is maintained both for citations and annotations. Subject keywords are simply a list of text strings separated by semicolons.

Note that all of this is metadata (that is, data about data). We have an academic discipline (such as anthropology) that has accumulated a good deal of data on its subject matter. Our citations are data about resources in which the field-specific data can be found. Our annotations are data about how that field-specific data can be taught, and our submissions are actually meta-metadata: who submitted a citation or annotation and when. Our topics are also meta-metadata: a controlled vocabulary of search categories for our repository of metadata.

Metadata is an important subject within XML. Metadata for XML data itself is essential both for the Semantic Web project and for the much-discussed Web services. Both of them aim at creating information structures that can be processed by machines as part of a purposeful action. Metadata standards are essential for that goal. In the case of Web services, you need a service description that will help you discover the service in the first place, in a repository that stores information about services. Once the service is discovered, you need another description or two to learn its specific functionality and how it is invoked. In the case of Semantic Web, you need a standard way of describing Web resources, groups of such resources, and relationships among resources. The standard metadata framework for this purpose is called *RDF* (*Resource Description Framework*). This is a major focus of W3C activity that has already resulted in several recommendations, with others in various stages of development.

RDF, Briefly

RDF is a big subject, but its basics are simple:

- The world consists of resources, each specified by a URI.
- Resources have properties, and each property has a value.
- Property values can be either resources or literal values (literals).

- An RDF document contains resource descriptions.
- Resource descriptions consist of statements that assign values to resource properties.

RDF statements are sometimes described as consisting of a subject (reference to the resource), a predicate (reference to the property), and the object (reference to the value). This terminology can be a bit confusing because it does not quite match typical English syntax. If we say "Trisha is the creator of the Web site `http://cs.colgate.edu`," "Trisha" is the subject of the English sentence but the object of an RDF description in which the resource is the Web site, the property is creator, and Trisha is the value of the property. The important thing for us is that RDF statements are resource-property-value triples.

RDF descriptions can be written in several different notations. The XML notation specified in the W3C recommendation *RDF Model and Syntax* is just one of them. (See `http://www.w3.org/TR/REC-rdf-syntax/`.) In that notation, the statement of the preceding paragraph will come out as

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://cs.colgate.edu">
    <author>Trisha</author>
  </rdf:Description>
</rdf:RDF>
```

Here, "author" is a property name that is not regulated by the RDF specification, whose only concern is the overall structure of description records. This structure is designed so that you can easily group together multiple statements about the same resource. For instance, if we wanted to add that Trisha worked together with Jonathan, and they did the work in 2001, we would simply add more children to the same `rdf:Description` element, as shown in Listing 7-1.

Listing 7-1. Example of RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://cs.colgate.edu">
    <author>Trisha</author>
    <author>Jonathan</author>
    <year>2001</year>
  </rdf:Description>
</rdf:RDF>
```

The namespace for the XML syntax of RDF descriptions defines a controlled vocabulary of element names that contains RDF (the root element) and Description. It does not define a controlled vocabulary of property names because RDF is intended to be a general framework that applies to many different subject domains, both on the Web and in the "real world." A vocabulary of properties that's appropriate for Web pages would not work well for bank accounts or online businesses. However, it does have significant overlaps with the vocabulary that's appropriate for describing books, and the librarians of the world were quick to understand that and to take the initiative in proposing a standard for such a vocabulary.

Dublin Core

As you may know, the world epicenter of online library science happens to be in Dublin, Ohio, at the headquarters of OCLC. The "O" in *OCLC* originally stood for *Ohio*, but, as the consortium grew to become national and then international in scope, its name mutated into *Online Computer Library Center*. Long before the Web, OCLC positioned itself as the leading authority on digital online catalogs and databases of bibliographic resources. When the Web developed into a huge library of online resources, OCLC was the first to propose a core descriptive vocabulary for Web pages. Because it resulted from a conference held in Dublin, Ohio, it was named the *Dublin Core Metadata Element Set*, or *Dublin Core* for short. Dublin Core predates XML: its first intended use (described in RFC 2731 *Encoding Dublin Core Metadata in HTML*, December 1999) was in META elements of HTML pages. Here is an example from the RFC:

```
<meta name    = "DC.Title" content = "The Communist Manifesto">
<meta name    = "DC.Creator" content = "Marx, K.">
<meta name    = "DC.Creator" content = "Engels, F.">
```

When XML came along, and especially after the first RDF recommendation, Dublin Core developed an independent existence as `dublincore.org`. Its membership comes from both W3C (especially its RDF group) and OCLC. It is moving rapidly to develop XML standards for digital library catalogs that would contain references both to traditional and Web resources. Just as W3C, the Dublin Core group is not a standards body and calls its documents "recommendations." It has adopted a procedure that is patterned after W3C's but with fewer intermediate stages, probably because Dublin Core recommendations are more modest in scope. Completed recommendations include the *Dublin Core Metadata Element Set* of fifteen standard elements (dcmes version 1.1), and a specification on how they are to be used within RDF descriptions (`http://dublincore.org/documents/2001/09/20/dcmes-xml/`, September 20,

2001). In the controlled vocabulary of dcmes-xml, our Listing 7-1 would come out as shown in Listing 7-2.

Listing 7-2. Example of Dublin Core RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:dc="http://purl.org/dc/elements/1.1/"
>
  <rdf:Description rdf:about="http://cs.colgate.edu">
    <dc:creator>Trisha</dc:creator>
    <dc:creator>Jonathan</dc:creator>
    <dc:date>2001</dc:date>
  </rdf:Description>
</rdf:RDF>
```

DCMES DTD

The *Dublin Core Metadata Element Set* (*DCMES*) specification comes with a DTD and an XML schema. We will not go through them in detail, but we will use a DTD excerpt to introduce all fifteen elements and the overall structure. All elements are collected in an entity:

```
<!ENTITY % dcmes "dc:title | dc:creator | dc:subject | dc:description |
dc:publisher | dc:contributor | dc:date | dc:type | dc:format |
dc:identifier | dc:source | dc:language |
dc:relation | dc:coverage | dc:rights" >
```

The entity is used to define the content model of `rdf:Description`. The definition says that all DC elements are optional and repeatable:

```
<!-- The resource description container element -->
<!ELEMENT rdf:Description (%dcmes;)* >
```

The rest of the DTD defines the fifteen elements. Each element is defined as PCDATA and has an optional `xml:lang` attribute. For example:

```
<!-- An entity responsible for making contributions
        to the content of the resource. -->
<!ELEMENT dc:contributor (#PCDATA)>
<!ATTLIST dc:contributor xml:lang CDATA #IMPLIED>
```

Some elements have an attribute that is, semantically, a resource:

```
<!-- An unambiguous reference to the resource within a given context. -->
<!ELEMENT dc:identifier (#PCDATA)>
<!ATTLIST dc:identifier xml:lang CDATA #IMPLIED>
<!ATTLIST dc:identifier rdf:resource CDATA #IMPLIED>
```

The identifier element can specify what it identifies as its `rdf:resource` attribute, but frequently it serves as an identifier for whatever is described by its parent `rdf:Description`. In our citation elements, we use it to provide traditional external identifiers, such as ISBNs, but we also use an identifier element to provide a system-generated unique identifier for each citation. This is an important structural feature of the overall application design: each citation, annotation, and submission element in the XML database has a unique system-generated identifier that is the first child of the element's record. We will return to this feature as we discuss the structure of the application.

Citation Elements

Our citation elements follow the dcmes DTD. Listing 7-3 provides an example.

Listing 7-3. Example of Citation

```
<rdf:Description>
  <dc:identifier>C0</dc:identifier>
  <dc:identifier>ISBN 0333776267</dc:identifier>
  <dc:title>Internet Ethics</dc:title>
  <dc:creator>Duncan Langford</dc:creator>
  <dc:format>Book</dc:format>
  <dc:publisher>ILRT, University of Bristol</dc:publisher>
  <dc:date>2000-06-06</dc:date>
</rdf:Description>
```

As discussed previously, every citation element has at least two identifier children. The first of them is a system-generated string that consists of the letter "C" (for *citations*) followed by a unique integer. The rest of the record is standard book metadata. The date is in the YYYY-MM-DD format, in conformance with the *XML Schema Part 2* specification.

Annotation Elements

Our annotations use XLink structures to relate annotations to what they annotate. XLink and RDF are in competition here: both propose vocabularies for describing graph structures. However, when the emphasis is on multiple links, the XLink vocabulary seems better suited for the task. It would also be possible to describe the annotations using RDF conventions, but the result would be much more verbose. Besides, we have already developed some expertise in XLink and a fair amount of code, so it makes sense to reuse them.

Each individual annotation is represented by an `ann` element. Just as with citations, the first child of each `ann` contains a unique ID that consists of the letter "A" followed by a unique integer. Because an `ann` is an extended link element, it has locator and arc children elements (that is, elements that have the `xlink:type` attribute equal to "locator" or "arc"). We use them in application-specific ways to support query processing, as explained after the example shown in Listing 7-4.

Listing 7-4. Example of Annotation

```
<ann xlink:type="extended">
<ident>A22</ident>
<elt xlink:type="locator" xlink:href="A22" xlink:label="this"/>
<elt xlink:type="locator" xlink:href="C4" xlink:label="Creole">
  <p>The title of the book is <em>Afro-Creole in the Caribbean</em></p>
</elt>
<elt xlink:type="locator" xlink:href="C5" xlink:label="Faces">
  <p>Faces of the Caribbean</p>
</elt>
<elt xlink:type="locator" xlink:href="C8" xlink:label="Lit">
  <p>Anthology of Caribbean literature</p>
</elt>
<elt xlink:type="arc" xlink:from="this" xlink:to="Creole">
  <p>Arc elements can contain XHTML descriptions also.</p>
</elt>
<elt xlink:type="arc" xlink:from="Faces" xlink:to="Creole">
  <p>There are interesting parallels between these two books.</p>
</elt>
</ann>
```

Several features of this element require an explanation. We provide them in the order of their appearance in the element.

The Content Model, Locator Elements, and `href` *Attributes*

The content model of `ann` is

```
<!ELEMENT ann (ident,elt*)>
```

where each `elt` is either a locator or an arc. Our locators locate *resource records* rather than resources themselves. The reason is that many of our resources are offline books and cannot be "located" by a URI. What this means is that, in our system, a locator's `href` attribute is a system-internal unique ID that locates a record, both within XML and within a relational database. An ID consists of a prefix (C, A, SC, SA) followed by an integer. Because the ID occupies a specific position within each element, it is easy to locate an element by its ID using XPath. The integer part of an ID also serves as the primary key of the record in the database, so it is easy to locate the record by its ID in the database also.

Descriptive Content

Both locator elements and arc elements can contain arbitrary XHTML to provide a description of the locator or the arc, as in this excerpt:

```
<elt xlink:type="locator" xlink:href="C4" xlink:label="Creole">
  <p>The book, whose complete title is <em>Afro-Creole in the Caribbean</em></p>
</elt>
```

Locator Labels

Most locator labels are constructed from book titles, but we also introduce a special "this" label to refer to the current annotation. This allows us to treat all annotations uniformly as extended links, which simplifies both XSLT and JSP/ASP code.

Schematically, instead of a simple link with an `href` attribute

```
<ann xlink:type="simple" href="someUri" />
```

we create an extended link that is processed in the same way as other extended links:

```
<ann xlink:type="extended">
<ident>A22</ident>
<elt xlink:href="A22" xlink:label="this" xlink:type="locator"/>
<elt xlink:href="someUri" xlink:label="someLabel" xlink:type="locator">
<elt xlink:from="this" xlink:to="someLabel" xlink:type="arc">
```

In addition, the “this” label greatly simplifies a recursive transitive-closure search for all resources (citations and annotations) that are linked to a given resource.

Submit Elements

Submission records, as we said, are meta-metadata: they record the date and authorship of a citation or annotation. RDF has heavy machinery for meta-metadata: there is a standard way of making a resource out of a statement. Because this operation makes a piece of description into a thing that we can talk about, logicians and RDF practitioners call it *reification*, which is Latin for “making into a thing.” Once a description is reified, it can be treated as any other resource that has properties.

We have decided not to follow that route because it would take us too deeply into RDF. Instead, we use a minimal structure adequate for our purposes, as shown in Listing 7-5.

Listing 7-5. Example of Submission Record

```
<submit>
  <ident>SA22</ident>
  <author>profProf</author>
  <date>2001-10-12 15:22:45</date>
</submit>
```

A submission record is a `submit` element with three children. The first child, as with citations and annotations, is a unique ID. The ID starts with the letter “S” followed by the ID of the corresponding citation or annotation. The remaining two children of a submission element contain information about the identity of the submitter and the date of the submission.

Some submission records have an additional fourth child to indicate that the corresponding resource has been committed. Recall that this application is designed for a team of cooperating adults who contribute educational materials to a shared database. Only members of the team have access to the database. All members of the team can view and add materials, but only a small subset of them, called *committers*, can commit the material to the tested and officially sanctioned version. We thus have two groups of users: developers and committers. (The terms are those used by most open-source software projects. See, for example, `http://www.apache.org`.)

Listing 7-6 provides an example of a committed record.

Listing 7-6. Example of Committed Record

```
<submit>
<ident>SC1</ident>
<author>profB</author>
<date>2001-10-17 14:54:31</date>
<committed by="admin" date="2001-10-22 14:55:18"/>
</submit>
```

Topics

Topics are keywords and phrases that are stored as a semicolon-delimited list of items within an attribute value.

The Structure of the Database

Now that we understand the structure of our XML data, we have to decide how to store it in a relational database. The first decision is easy: for each of our three kinds of structured data (citations, annotations, and submission records), we will have three tables, with each record in them corresponding to an XML element. Note that each table will have a primary key, and all joins will be on primary keys. This guarantees efficiency because virtually all DBMS build indexes on primary keys, and joins on primary keys take constant time.

The next question is how to store an element. One can take either of two extreme approaches. At one extreme, an XML document is stored in its serialized form in a single text field. At the other extreme, each descendant of the element is stored as a separate field. However, there are any number of compromise solutions, in which some descendants (those that are likely to be used in queries) are unrolled into separate fields while the rest (or perhaps the entire element) are stored in the serialized form.

In our database, we will completely unroll submission records because they have a flat structure with a small and fixed number of children, but adopt a compromise solution for citations and annotations: some of the descendants will be stored in separate fields, but there will also be a field to contain the entire serialized element. This will make it possible to search citations and annotations by, for example, `dc:creator` and further filter the retrieved records by an XPath condition. (See the later section on refset operations.)

Another common problem is how to store descendants that can be repeated any number of times. A common solution, adopted here, is to store their contents in a delimited text string and break that string into individual components as needed. In our database, the delimiter is the end-of-line marker. The same delimiter is used in the user interface: an entry field for, say, authors, is a text area, and

the user is instructed to enter each author (if there's more than one) on a separate line. Eventually, the end-of-lines entered by the user end up in the database field.

In summary, the database (xmlp.mdb) consists of four tables, corresponding to the main divisions in the data: `rdbCitations`, `rdbAnnotations`, `rdbSubmissions`, and `rdbTopics`.

The Structure of the Application

The application consists of the following major components:

- user interface: Web pages (JSPs) with forms and XSLTs that go with them
- XML repository and persistent store
- query result set modifiable by follow-up queries (We call it *reference set*, abbreviated as *refset*.)

Supported Actions

The application supports two kinds of actions that broadly correspond to SELECT and UPDATE queries of the relational database. We call them *search actions* and *update actions*. Some actions are available to all users, and others are available only to a special category of users (the committers).

> **NOTE** *In this simplified version of the application, we distinguish only two categories of participants: users and committers. In the production version, we distinguish three categories: users, developers, and committers. Users have read-only access to committed material and do not need to log in. Developers are like users, but they can also submit new materials and have write access to materials of their own. Committers are like developers, but they can also change the status of submitted materials to committed after an editorial process and the developer's consent. The extra complications involved are of little value to the reader of this book.*

Search actions operate on a collection of references to objects in the database ("refset" for short). The actions are add, delete, clear, and display. (See Figure 7-1.) If the action is "add," then the XML data store is searched for matching records, and they are added to the refset; if the action is "delete," the current refset is searched for matching records, and they are removed from the refset. The "clear" action empties the refset, and the "display" action (the default) shows

its contents in a text area. None of the refset actions have any effect on the contents of the database.

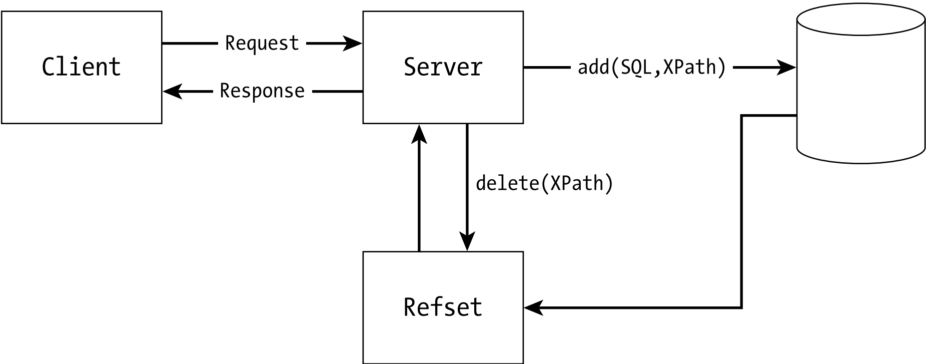

Figure 7-1. Search actions: client, server, database, and refset

Within update actions, all users can add a citation or an annotation to the database. For committers, there is also a form to change the status of a record to "committed," and another form to add a new subject keyword or phrase. Update actions are summarized in Figure 7-2.

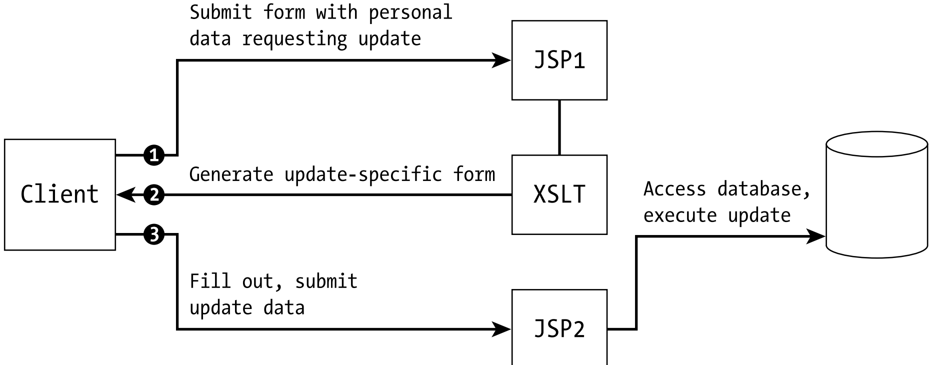

Figure 7-2. Update actions: client, server, and database

Application Components and Files

Code files for the application fall into four groups: top-level files, action files, user-interface files, and included files.

Top-level files are index.jsp (for login), logout.jsp, error.jsp, and rdbFrames.jsp.

Action files and user-interface files are as follows:

- rdbCtl.jsp is the initial source for the control frame. It contains code for reloading the data. Like all code files that use the database, it uses (by inclusion) getConn.jsp.
- rdbRefSetOps.jsp performs all refset-related actions. It uses getRef.jsp, getRefs.jsp, and (by inclusion) getRefBase.jsp.
- Included files are getRefBase.jsp and getConn.jsp.
- upAddCite.jsp, upAddAnnot.jsp, and upCommit.jsp are for update actions.
- uiCitation.jsp, uiCitation.xsl, uiAnnotation.jsp, and uiAnnotation.xsl are user-interface files that display appropriate forms. The action of the uiCitation.jsp form is upAddCite.jsp, and similarly for uiAnnotation.jsp and upAddAnnot.jsp.

In outline, most user interactions with the system follow the scenario detailed in Table 7-1.

Table 7-1. Typical Scenario of Use

USER	APPLICATION
Log in	Verify login; store login info in session cache; display top form, rdbCtl.jsp, with a menu of actions
Select action	Display appropriate form
Fill out form and submit	Perform action; display result in data frame; display follow-up form, if any, or return to top form

The Login Page, the Frames Page, and Access Control

The login page (Listing 7-7) collects the username and password and submits them to the application.

Listing 7-7. The Login Page, index.jsp in the rdb Directory

```
<%@ page  errorPage="error.jsp"  %>
<html><head><title>XML Repository</title></head>
<body>
Welcome to the XML Repository. Please enter username and password.
<form action="rdbFrames.jsp" method="post" target="_top">
Name:<input type="text" name="unm" value="admin" size="20"/>
<select size="1" onchange="with(this.form)unm.value=this.value">
 <option selected="true" value="admin">committer</option>
 <option value="anyName">user</option>
</select>
<br/>
Pwd:<input type="password" name="upw" value="pwd" size="30"/>
<input type="submit" value="login"/>
</form>
<!-- the next line is in case your login page appears in a frame -->
<script language="JavaScript">
if(window != window.top)
  document.write('<a href="index.jsp" target="_top">Get out of frames!</a>');
</script>
</body></html>
```

The frames page displays two frames: a control frame with a menu of available actions and a data display frame (which is initially empty). However, before the frames are displayed, the page checks to see that the username variable is not empty. All the remaining pages begin with this check, in case a casual or malicious visitor accessed the page without going through the login process.

In the code shown in Listing 7-8, login validation is primitive: the password is the same for everybody, and, to be a committer, you only have to log in with the "admin" username. These are, of course, placeholders for a table lookup on a secure server. The main point for us is that, as a result of this procedure, we store the username and the `isCommitter` string value in the session cache so that other pages in the session have access to them. (We use a string rather than a Boolean `isCommitter` because this simplifies Java code in a minor way.)

Listing 7-8. The Frames Page, `rdbFrames.jsp`

```
<%@ page  errorPage="error.jsp"
%><jsp:useBean id="sessCache" class="java.util.Hashtable" scope="session"
/><%
  String unm=(String)sessCache.get("unm");
  if(unm==null) { // no username yet
    unm=request.getParameter("unm");
    if(null!=unm)sessCache.put("unm",unm);
```

```
    String pwd=request.getParameter("upw");
    boolean pwdOk="pwd".equals(pwd); // a place holder
    if(!pwdOk || unm==null || unm.length()==0)
     { sessCache.remove("unm"); %><jsp:forward page="index.jsp"/><% }
  String isCommitter= "no"; // a
  if("admin".equals(unm))isCommitter="yes"; // a place holder
  sessCache.put("isCommitter",isCommitter);
/* we get this far only after password validation */
%><html><head><title> Frames for Vector DB </title></head>
<frameset cols="45%,55%">
  <frame name="ctlFrame" src="rdbCtl.jsp">
  <frame name="dataFrame" src="about:blank">
</frameset></html>
```

At this point, the screen looks as shown in Figure 7-3.

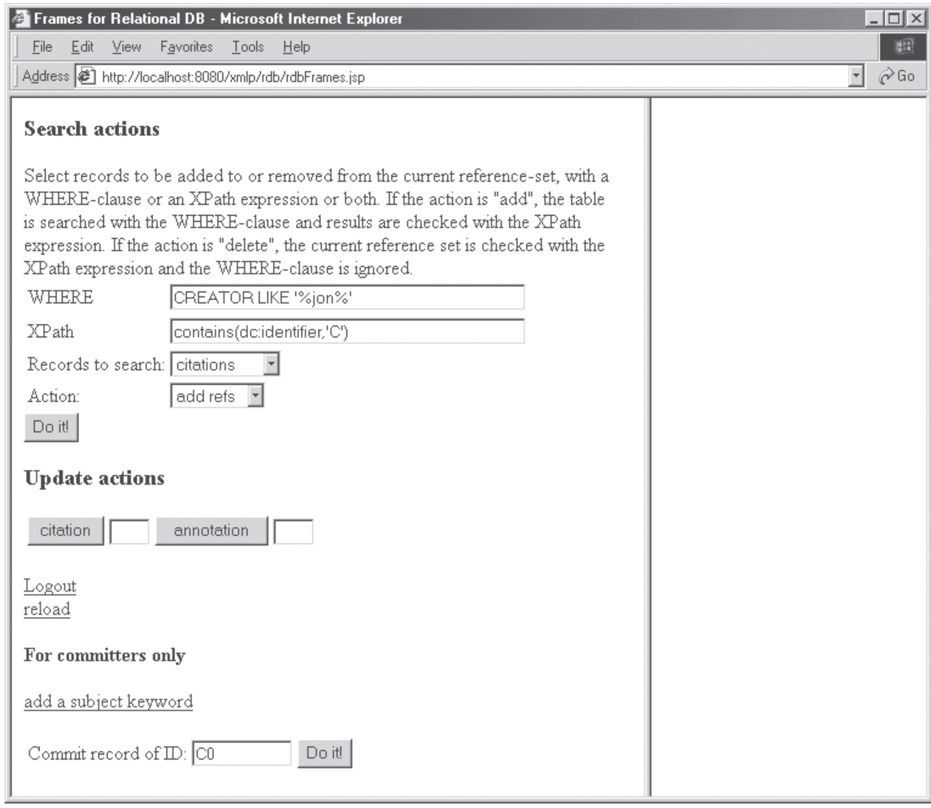

Figure 7-3. The frames page with top form in the control frame

The page displayed in the control frame, rdbCtl.jsp, contains both an HTML form and a good deal of Java code.

The Top Control Page, rdbCtl.jsp

In outline, the page proceeds as follows:

1. imports and beans
2. login check
3. initialize and reload
4. the HTML form
5. code to support initialization and reloading

The last item contains code related to database access. When we reach that point, we will interrupt to review the main concepts and how they are implemented in Java.

Imports and Beans, Login Check

Imports are extensive and listed individually, without any wildcards. They come from the following sources:

- DOM and SAX
- JAXP packages for parsing and transformation
- Java sql package for database access
- Java servlet classes
- other Java
- Apache-specific class for XML processing

Next, we instantiate three beans: the application cache, the session cache, and a hashtable to hold the refset.

```
<jsp:useBean id="appCache" class="java.util.Hashtable" scope="application"/>
<jsp:useBean id="sessCache" class="java.util.Hashtable" scope="session"/>
<jsp:useBean id="refSet" class="java.util.Hashtable" scope="session"/>
```

The caches are used for application and session data as follows.

- appCache holds basic XML information (namespaces) and basic database information, including the number of records in the database. As a minor optimization, it also holds the path to the Web application's root, so we don't have to recalculate it every time we need to resolve a reference.
- sessCache holds several session-level variables, including a parser factory to get an XML parser, a transformer to serialize query results, and a string showing whether the session's user is a committer. (We use a string rather than a Boolean to simplify Java code in a minor way.)

You will see how the caches are used momentarily, but first we have to check the login. Recall (from `frames.jsp`, Listing 7-8) that, if login is successful, the username is stored in the session cache. Now we only have to check the session cache to see if the username is there:

```
String unm=(String) sessCache.get("unm");
if(unm==null)
  { %>
      <jsp:forward page="index.jsp"/>
    <%
  } // not yet logged in.
```

From this point on, we know that the user is logged in; next we want to find out whether she is a committer:

```
String isCommitterStr=(String) sessCache.get("isCommitter");
boolean isCommitter= "yes".equals(isCommitterStr);
```

We can now proceed to initialize the session and, perhaps, (re)load the entire application.

Session Initialization and Application (Re)Loading

We proceed to initialize session variables, as shown in Listing 7-9.

Listing 7-9. Session Variables

```
// get from cache or create new and store in cache
DocumentBuilderFactory dbf=getDBF(sessCache);
DateFormat dateFormat=getDateFormat(sessCache);
Transformer trans=getTransformer(sessCache);
```

These three function calls all follow the same logic: see if the variable is in session cache; if not, create a new instance, store it in cache, and return. Listing 7-10 shows `getDBF()`, declared later in the file.

Listing 7-10. `getDBF()`

```
public DocumentBuilderFactory getDBF(Hashtable sessCache){
  DocumentBuilderFactory dbf=(DocumentBuilderFactory)sessCache.get("dbf");
  if(dbf==null){
    dbf=DocumentBuilderFactory.newInstance();
    dbf.setNamespaceAware(true);
    dbf.setValidating(false);
    sessCache.put("dbf",dbf);
  }
  return dbf;
}
```

Next we retrieve two variables from the request object to see whether the user wants to do something other than run queries:

```
boolean reload= "yes".equals(request.getParameter("reload"));
String newTopic = request.getParameter("newTopic");
```

If there is a new topic to add (an option available only to committers, as you will see shortly), then we add it both to the database and to the application cache:

```
if(newTopic != null && newTopic.length() > 0)
  addTopic(appCache,newTopic);
```

We will show the code for `addTopic()` after we cover JDBC. In the meantime, if the user wants to reload, or if this is the very first time the application is run, then we load or reload the application.

(Re)Loading the Application

The first thing to note is that we don't want to start the process of (re)loading if other users are running the program, and we don't want another user to invoke the program while it is being (re)loaded. In databases, this kind of blocking is performed by the database management system. Java has a special construct that ensures that only one "thread of execution" can access a certain variable. The construct uses the SYNCHRONIZED keyword. We synchronize on the built-in `application` object, which is unique to each JSP application. (See Listing 7-11.)

Listing 7-11. Synchronized Reloading of the Application

```
synchronized(application){
  if(null==appCache.get("idCount") // no records yet
      || reload){ // user asks to reload
    appCache.clear();
    refSet.clear();
    setAppCacheRootElt(dbf, appCache); // basic XML info, e.g. namespaces
    setAppCacheDB(appCache);       // basic DB info, e.g. driver name
    getWebAppPath(appCache,request); // path to web apps root
  }
}
```

The HTML Form

Next in rdbCtl.jsp is the HTML form that is displayed in the control frame. The only Java code in it is the conditional to check whether the user is a committer, in which case the section "For committers only" is also displayed:

```
<% if(isCommitter) { %>
// display widgets to add a keyword or commit a record
<h4>For committers only</h4>
  <a href="uiTopic.jsp">add a subject keyword</a><br/><br/>
  <form action="commit.jsp" target="dataFrame">
    <table>
    <tr>
      <td>Commit record of ID:</td>
      <td><input name="ref" type="text" size="10" value="CO"></td>
     <td><input type="submit" value="Do it!"></td>
     </tr>
    </table>
  </form>
<% } %>
```

Apart from this little piece of scripting, the form's code is trivial and can be easily reconstructed from the screen shot in Figure 7-3.

Definitions of Java Methods

The initialization code in the preceding sections makes a number of function calls, using functions declared later in the JSP. You have seen one of those functions, `getDBF()` (Listing 7-10). It is time to review them all and give a more representative sample. The functions fall into three groups:

- Functions similar to `getDBF()` that try to retrieve a session object from cache, and, in case of failure, create a new such object and store it in cache: `getTransformer()`, `getDateFormat()`, and `getWebAppPath()`.
- Functions used in (re)loading the application: `setAppCacheRootElt()`, `setConnectionData()`, and `setAppCacheDB()`. We will review the first one in this section and the last two after covering JDBC.
- The `addTopic()` function that implements the "add topic" action. Unlike all other actions, this one doesn't have a JSP of its own because it takes very little code. It does rely on the included `getConn.jsp` to obtain a connection to the database. Like all other database-related material, it will be presented after the JDBC section.

`getTransformer()`, `getDateFormat()`, *and* `getWebAppPath()`

Listing 7-12 displays the three functions listed in the first bullet. As explained, they all follow the same pattern as `getDBF()` of Listing 7-10.

Listing 7-12. Transformer, DateFormat, and WebAppPath

```
public Transformer getTransformer(Hashtable sessCache)
  throws TransformerConfigurationException {
  Transformer trans=(Transformer)sessCache.get("trans");
  if(trans==null){
    TransformerFactory tFactory = TransformerFactory.newInstance();
    trans = tFactory.newTransformer();
    trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,"yes");
    trans.setOutputProperty(OutputKeys.INDENT,"yes");
   sessCache.put("trans",trans);
  }
  return trans;
}
```

```
public DateFormat getDateFormat(Hashtable sessCache){
  DateFormat dF=(DateFormat)sessCache.get("dateFormat");
  if(dF==null) dF=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
  sessCache.put("dateFormat",dF);
  return dF;
}

public String getWebAppPath(Hashtable appCache,HttpServletRequest request)
  throws Exception{  // returns, e.g., "http://localhost:8080/xmlp/rdb/"
    String webAppPath=(String)appCache.get("webAppPath");
    if(webAppPath==null){
      String myURI=HttpUtils.getRequestURL(request).toString();
      webAppPath=myURI.substring(0,1+myURI.lastIndexOf('/'));
      appCache.put("webAppPath",webAppPath);
      }
    return webAppPath;
  }
```

setAppCacheRootElt()

The only remaining function that does not deal with database access is `setAppCacheRootElt()`. Its task is a bit peculiar, reflecting the peculiar way in which our repository operates: we have a great number of XML elements (citations and annotations) stored in a relational database, but we don't have a root element within a document that contains them all. The elements stored in the database all use namespace prefixes for RDF, Dublin Core, and XLink, but the namespaces are not declared anywhere. `setAppCacheRootElt()` sets up a purely abstract "root element" for all the citations and annotations in the database, declares namespaces on it, and stores it in the application cache. (See Listing 7-13.) When namespace resolution is needed, the declarations are retrieved and prefix mappings are processed. The processing is done here, in an Apache-specific way, by an `org.apache.xml.utils.PrefixResolverDefault` object that implements the `PrefixResolver` interface.

Listing 7-13. `setAppCacheRootElt()`

```
public void setAppCacheRootElt(DocumentBuilderFactory dbf, Hashtable appCache)
    throws Exception{
  String xmlStr=
   "<dbDataRoot xmlns:dc='http://purl.org/dc/elements/1.1/' "
    +" xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#' "
    +" xmlns:xlink='http://www.w3.org/1999/xlink'/>";
  DocumentBuilder docBuilder=dbf.newDocumentBuilder();
```

```
  Document doc=docBuilder.parse(new InputSource(new StringReader(xmlStr)));
  Element rootElt=doc.getDocumentElement();
  appCache.put("namespaces",new PrefixResolverDefault(rootElt));
}
```

The remaining definitions all have to do with JDBC and will be covered in and after the JDBC section:

```
public void setConnectionData(Hashtable appCache)
public void setAppCacheDB(Hashtable appCache)
public void addTopic(Hashtable appCache,String newTopic)
```

This concludes rdbCtl.jsp. To remind you where we are in a typical use case, the user has logged in and is ready to select an action to perform. Apart from reload (covered in this section) and logout, all of them involve database access. Our next task is to review the general concepts of database access from a computer program and how they are implemented in Java.

Driver, Database, Connection, and Statement

Java libraries for database access are collectively known as *JDBC* (*Java Database Connectivity*). They define a number of classes and interfaces that create a vendor-independent database connectivity layer. The key to vendor independence is a JDBC driver, a software package produced by database or third-party vendors that encapsulates DBMS-specific features. (For instance, it converts between DBMS-specific data types and JDBC data types.) A database featuring more than 150 JDBC drivers (some of them free, others commercial products) is available at `http://industry.java.sun.com/products/jdbc/drivers`.

In addition to database-specific drivers, Sun provides, as part of the standard Java distribution, a JDBC–ODBC bridge that makes it possible to connect to ODBC data sources using JDBC APIs. This is the driver that we use in our code. However, we store the name of the driver in the application cache, and, to use a different driver, you only have to change that name in the cache (after you have installed your driver, as explained in the next section).

Installing and Using a Driver

JDBC drivers are usually distributed as JAR (Java archive) files. To install, simply add the JAR file to your classlist. To use, include a line like this in your code:

```
Class.forName(your--driver-name);
```

If the JDBC–ODBC bridge is used, this line comes out as

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

To encapsulate this code, we store the name of the driver in the application cache, keyed by the "dbDriver" string, and so the driver is instantiated by this line of code:

```
Class.forName(appCache.get("dbDriver").toString());
```

Once the driver is instantiated, you can proceed to use JDBC code. A typical scenario proceeds as follows:

1. Obtain a connection, typically from a connection pool.
2. Create a statement object, to execute SQL statements.
3. Run a database session, executing statements, processing result sets, and so on.
4. Close connection (or return it to the connection pool).

The following subsections explain these steps and illustrate them with examples from our code. The first step is to obtain a connection to your database.

Connections and Connection Pooling

The simplest way to obtain a connection is via a public static `getConnection()` method of the DriverManager class. The method takes three parameters: the database URI, username, and password. The database URI must be in the format specified in the documentation of your JDBC driver. In the case of the JDBC–ODBC bridge, you must create a DSN (Data Source Name) using the ODBC manager and use that name as the database URI. Assuming that the DSN is "xmlp" and the username and password can be empty strings, you would obtain a connection by this line of code:

```
Connection con=DriverManager.getConnection("jdbc:odbc:xmlp","","");
```

We wrap this action in two layers of indirection to make the code more general. First, just as with the driver name, we store the database URI in the application cache. This is done in one of the methods of rdbCtl.jsp that we skipped on first reading, as shown in Listing 7-14.

Listing 7-14. Store JDBC Driver Name and Database URL in Application Cache

```
public void setConnectionData(Hashtable appCache){
  appCache.put("dbDriver","sun.jdbc.odbc.JdbcOdbcDriver");
  appCache.put("dbURL","jdbc:odbc:Xmlp");
}
```

Second, we place the code that obtains a connection into a `getConnection()` function defined in the included getConn.jsp. Therefore, in our application, a connection is obtained by

```
con=getConnection(appCache);
```

where getConnection() is as shown in Listing 7-15.

Listing 7-15. Code for `getConnection()` *from getConn.jsp*

```
public Connection getConnection(Hashtable appCache){
 try{ // getConnection could retrieve from connection pool
  Class.forName(appCache.get("dbDriver").toString());
  return DriverManager. // statement continued on next line
          getConnection(appCache.get("dbURL").toString(),"rdb","pwd");
 }catch(Exception ex){
   System.err.println("failed to get DB connection: "+ex);
   return null;
 }
}
```

This makes it easy to replace our simple code with a more sophisticated method of managing connections called "connection pooling". The motivation for connection pooling is that obtaining a connection is a computationally expensive operation that should be done as rarely as possible. The idea of connection pooling is simple enough: the application obtains a pool of connections in one action, typically at startup, and asks for another connection only when the pool dries up. When a new user asks for a connection, it is allocated from the pool, and, when a user releases a connection, it is returned to the pool. In our code, releasing a connection is also encapsulated as a method, `freeConnection()`, that can be rewritten to use connection pooling:

```
public void freeConnection(Connection con,Hashtable appCache){
 try{ // freeConnection could return con to a connection pool
  if(con!=null)con.close();
 }catch(Exception ex){
   System.err.println("failed freeConnection: "+ex);
 }
}
```

Several connection pool implementations are available, and the latest version of JDBC implements a standard API for managing a connection pool. To incorporate connection pooling in our application, you would only have to install the appropriate software and rewrite our `getConnection()` and `freeConnection()`, both in getConn.jsp.

Statement and ResultSet

Once you have a connection, you can run SQL statements. SQL contains both a data definition language that defines the structure of the database and a data manipulation language; data manipulation language, in turn, contains SELECT queries that only retrieve data and UPDATE queries that modify the data (but not the structure of the database). We do provide an example of data definition language in rdbCreateTables.jsp (so that a database to use with our code can be created even if you don't have a copy of MS Access on your machine), but the application proper uses only SELECT and UPDATE queries. The simplest way to use them is to create a Statement object and run its `executeQuery()` method for SELECT queries and `executeUpdate()` method for UPDATE queries. Both methods take a string argument that is a well-formed SQL query.

If you run an UPDATE query, the returned value is an integer—the number of rows that have been inserted, deleted, or modified. (This value is frequently ignored.) For an example of `executeUpdate()`, consider `addTopic()` from rdbCtl.jsp. (See Listing 7-16.) It adds a new topic word or phrase to the database table that holds them, and also to the appCache, where topics are held as a semicolon-separated string.

Listing 7-16. `addTopic()` *from rdbCtl.jsp*

```
public void addTopic(Hashtable appCache,String newTopic) {
  Connection con=null;
  Statement stmt=null;
  try{
      con=getConnection(appCache); // our method from getConn.jsp
      stmt = con.createStatement();
// construct SQL query; the new topic has to be quoted in it,
//  hence single quotes within double quotes
      String sqlStr="INSERT INTO rdbTopicNames VALUES ('"+newTopic+"')";
// insert new topic into database;
// returned integer is ignored but possible error is caught
      stmt.executeUpdate(sqlStr);
// insert new topic into appCache
      String topicNames=appCache.get("topicNames").toString();
      if(topicNames==null) topicNames="";
```

```
        appCache.put("topicNames",topicNames+newTopic+"; ");
      }catch(Exception ex){ex.printStackTrace();}
     finally{
       if(stmt!=null) try{stmt.close();stmt=null;} catch(Exception ex){}
       freeConnection(con,appCache); // our method in getConn.jsp
      }
}
```

If you run a SELECT query, the returned value is a ResultSet object that provides sequential access to the returned records. Much like an enumeration or a tree traverser, it has a `next()` method that returns the next record or null if we have reached the end of the record set. Within each record, individual fields are retrieved by `getXX()` methods, where *XX* stands for a data type: `getInt()`, `getString()`, and so on. The argument to all these methods is either an integer giving the number of the field in the record (beginning with 1) or the field's name in the database table.

For an example of a SELECT query, consider `setAppCacheDB()`, the final leftover from rdbCtl.jsp. It retrieves from the database and stores in appCache two items: the number of records in the database (as `idCount`) and the semicolon-separated list of topics:

```
public void setAppCacheDB (Hashtable appCache){
      Connection con=null;
      Statement stmt=null;
      ResultSet rs=null;
      int idCount=0;  StringBuffer topics=new StringBuffer();
     try{
       setConnectionData(appCache); // see Listing 7-14
       con=getConnection(appCache); // see Listing 7-15
       if(con==null) return;
       stmt = con.createStatement();
// retrieve the last record from the rdbSubmissions table
// the number of submissions is the number of records in database
       rs=stmt.executeQuery("SELECT 1+Max(ident) FROM rdbSubmissions");
       if(rs.next())  // if result set not empty
         idCount=rs.getInt(1); // retrieve first field of record, which is the
       rs.close();
       rs=stmt.executeQuery("SELECT * FROM rdbTopicNames");
       while(rs.next())topics.append(rs.getString(1)+"; ");
     }catch(Exception ex){ex.printStackTrace();}
    finally{
      if(rs!=null)try{rs.close();}catch(Exception ex){}
      if(stmt!=null)try{stmt.close();}catch(Exception ex){}
```

```
     freeConnection(con,appCache); // in getConn.jsp; similar to getConnection()
// store two items in appCache
     appCache.put("idCount",new Integer(idCount));
     appCache.put("topicNames",topics.toString());
    }
}
```

PreparedStatement

In many situations, a better alternative to Statement is the JDBC PreparedStatement. PreparedStatement is created with an SQL query imprinted on it at construction. It is more efficient than plain Statement because its SQL query is compiled once and can be reused many times with different parameters.

Consider a simple example: suppose you have a database table that has names and email addresses and you want to be able to retrieve an address (or addresses) by name. You have created a database connection as described in the preceding sections, and you are ready to create a query string and a statement object. Assume that the name to search by is in the `currentName` variable, probably retrieved from the Request object. With plain Statement, you would create a query string like this:

```
"SELECT addr FROM addrBook WHERE name='" + currentName + "'"
```

Note that you need single quotes within double quotes so that the value of `currentName` comes out quoted in the resulting string. This is error prone and may result in nasty complications: what if the name has special characters, as in “O’Donnell”? Besides, you have to remember to quote strings but not integers or dates, except if you insert dates as strings. The PreparedStatement, in addition to being more efficient, provides a simple and uniform way of filling in arbitrary parameters, as follows:

First, you create a query string differently, with question marks as placeholders for parameters to be filled in, and you use the query string in creating your PreparedStatement:

```
String queryStr = "SELECT addr FROM addrBook WHERE name=?";
PreparedStatement prepStmt = conn.prepareStatement(queryStr);
```

Next, you fill in the value of the parameter using one of many data type-specific procedures that are provided for that purpose. In this case, we need `setString()`:

```
// set the value of the first parameter of PreparedStatement to currentName
prepStmt.setString(1,currentName);
// run the query
ResultSet rs= prepStmt.executeQuery();
```

Note that `executeQuery()` does not take any arguments because the query string is already imprinted on the PreparedStatement and its parameter is already set. For UPDATE queries, you would use `executeUpdate()` rather than `executeQuery()`, as you will see in several rdb query implementations in the next section.

In addition to `setString()`, PreparedStatement has `setInt()`, `setBlob()`, `setBoolean()`, `setDate()`, and so on, even including `setObject()` in which we provide an arbitrary object and tell the database what standard SQL type to treat it as. Here's a sample from upAddCite.jsp, presented in full in the next section:

```
pStmt.setInt(1,identInt);
pStmt.setString(2,identifierStr);
. . .
pStmt.setObject(7,xmlValue,java.sql.Types.LONGVARCHAR);
```

PreparedStatement is the last remaining JDBC feature that is used in the application of this chapter. Many more features are available for more-advanced use, such as connection pooling, access to metadata, transactions, and stored procedures.

JDBC Equivalents in ASP

Database access in ASP differs in minor details from JDBC but operates with the same concepts and similar objects. For an example, consider an ASP that accesses our submissions table (rdbSubmissions), retrieves submission records by their author, and outputs them as an HTML table:

```
<%@ LANGUAGE="VBSCRIPT" %>
<% Option Explicit %>
<html><head><title>rdbSubmissions author </title>
<META HTTP-EQUIV="Content-Type" content="text/html; charset=iso-8859-1">
</head><body>
<table border="1">
```

```
<%
Dim oRS,intCount,oField,author,query  ' declare variables
'create a RecordSet object using ADODB (Active Data Object)
Set oRS=Server.CreateObject("ADODB.RecordSet")
'get data from Request
author = Request.QueryString("author")
'create SQL query string
query = "SELECT * FROM rdbSubmissions WHERE Author LIKE '" & author & "%'"
'open connection to database and run query in a single Open method of RecordSet
oRS.Open query,"DSN=xmlp;UID=;PWD="
Response.Write "<tr>"
'output field names as HTML table headers
For Each oField in oRS.Fields
   Response.Write "<th>" & oField.Name & "</th>"
Next
Response.Write "</tr>"
'output RecordSet, a record per row of HTML table
Do While NOT oRS.EOF   'same syntax as in reading a file
  Response.Write "<tr>"
  For intCount = 0 to oRS.Fields.Count-1
    Response.Write "<td>" & oRS.Fields(intCount).Value & "</td>"
  Next
  Response.Write "</tr>"
  oRS.MoveNext
Loop
%>
</tr></td></table></body>
</HTML>
```

In this version, a connection object is created behind the scenes as part of populating a RecordSet object with query results, but it is also possible to go in the way JDBC requires, with an explicit connection object within which we use a query to create a ResultSet object.

This particular ASP doesn't mention XML at all, but its output is legal XML (XHTML) and can be processed using MSXML, as in other examples in the book. We used rdbSubmissions so that you can read through it once as a plain ODBC/ASP/HTML construction without thinking about XML, but, if you use the other tables and retrieve the xmlValue field, the MSXML DOMDocument constructions can parse the string and do whatever you like with it, just as Java's DocumentBuilder can.

The .NET framework has an entirely new set of tools for working with relational data and converting it to XML, but this is a subject for another book, or for the next edition of this one.

Query Implementations 1: UPDATE Queries

Recall that our application's functionality mostly consists of two kinds of queries: UPDATE queries that add new records to the database or modify existing ones, and SELECT queries that retrieve data from the database into a data structure called *refset*. A refset can be further modified by either removing some data from it or adding more data from the database.

In terms of code, the application consists of top-level files, included files, action files, and user-interface files:

- Top-level files are index.jsp (for login), logout.jsp, error.jsp, and rdbFrames.jsp.
- Included files are getConn.jsp and getRefBase.jsp.
- User interface files are uiCitation.jsp, uiCitation.xsl, uiAnnotation.jsp, and uiAnnotation.xsl. The JSPs use the corresponding XSLTs to display appropriate forms.
- Action files for update queries are upAddCite.jsp (the action of the uiCitation.jsp form) and upAddAnnot.jsp (the action for the uiAnnotation.jsp form). In addition, there is upCommit.jsp for changing the status of a record to committed.
- The action file for refset operations is rdbRefSetOps.jsp, which performs all refset-related actions. It uses getRef.jsp, getRefs.jsp, and (by inclusion) getRefBase.jsp.
- Finally, rdbCtl.jsp is the initial source for the control frame. It contains code for reloading the data. Like all code files that use the database, it uses (by inclusion) getConn.jsp.

We have covered all the code of rdbCtl.jsp, most of it in the section "Structure of the Application" and the rest as examples in the JDBC section. In the remainder of the chapter, we are going to show a representative sample of action files. In particular, for UPDATE queries, we will work through the user interface files for adding a citation and the upAddCite.jsp that performs that operation. For refset operations, we will show rdbRefSetOps.jsp. (The user interface for refset operations is in rdbCtl.jsp.)

UPDATE Queries Overview

The progression of steps in creating or modifying a citation is as follows:

- From rdbCtl.jsp, the user fills in the Add Citation form and submits. The action of the form is uiCitation.jsp.
- uiCitation.jsp constructs an input form for citations using uiCitation.xsl and information about the user (committer or not) stored in sessCache. The constructed form is displayed in the control frame. The action of the form is upAddCite.jsp.
- upAddCite.jsp constructs a citation element from request data submitted by the user, stores it in the database according to the adopted scheme, and also displays it in a text area for the user to review. Each time we add a citation or an annotation, we add a submission record as well.

We will follow this process through uiCitation.jsp, uiCitation.xsl, and upAddCite.jsp.

uiCitation.jsp

This JSP offers different functionality to committers and general users (noncommitters). General users can only add new citations. They will be presented with a blank form (generated by uiCitation.xsl) to enter citation data; if they enter an existing citation's ID into the appropriate box, it will be ignored.

Committers can also add new citations, but they can also edit existing ones. To do that, they enter an existing citation's ID into the appropriate box, and the system will display the existing record for the committer to edit and save. (Typically, to find out the ID of an existing record, the committer would use a SELECT query first.)

We will display the citation entry form in the next section, together with the XSLT code that generates it. The code of uiCitation.jsp proceeds as follows:

1. Start with the familiar imports, beans, and the login check. A new element in this section is the include directive to include getRefBase.jsp: we are going to use its `getXmlValue()` method to retrieve a serialized citation element from the database.
2. Construct the XSL filename, which is the same as the JSP's name but with the .xsl extension.

3. Check the ID: if present and valid, retrieve the XML record from the database for input to XSLT; otherwise, provide an empty element for input.

4. Construct a transformer for XSLT and run it on XML input. Send output to the browser.

5. Finish up with links to logout and rdbCtl.jsp.

The code in Listing 7-17 is divided by highlighted comments that correspond to the items in this outline.

Listing 7-17. The Code of uiCitation.jsp

```
<%@ page errorPage="error.jsp"
    import="javax.xml.transform.Transformer,
            javax.xml.transform.TransformerFactory,
            javax.xml.transform.stream.StreamSource,
            javax.xml.transform.stream.StreamResult,
            java.io.StringReader,java.io.File,
            java.sql.Connection,java.sql.Statement,
            java.sql.ResultSet,java.sql.DriverManager,
            java.util.Hashtable"
%><jsp:useBean id="appCache" class="java.util.Hashtable" scope="application"
/><jsp:useBean id="sessCache" class="java.util.Hashtable" scope="session"
/><%@ include file="getRefBase.jsp" %><%
String unm=(String)sessCache.get("unm");
if(unm==null)
  { %><jsp:forward page="index.jsp"/><% } // not yet logged in.
boolean isCommitter= "yes".equals((String)sessCache.get("isCommitter") );
// Start HTML output
%><html><head><title>Create Citation</title></head>
<body>
On this page, you can create a new citation.
<% if(isCommitter){ %>
You can also enter an existing citation's id and new values for its fields.
<% } %>
<%
  // Construct the XSL file name: same as this JSP, but with .xsl suffix
  String pathToJSP=application.getRealPath(request.getServletPath());
  String pathToXSL=pathToJSP.substring(0,1+pathToJSP.lastIndexOf('.'))+"xsl";
  String xslUri=new File(pathToXSL).toURL().toString();
  // check ID; construct XML input to Transformer
  String idStr=request.getParameter("idStr");
  if(idStr==null || !idStr.startsWith("C"))idStr="";
```

```
  String citeRecStr=null;
  if(idStr.length()>0) // valid ID; retrieve record from database
    citeRecStr=getXmlValue(idStr,appCache);
  if(citeRecStr==null ||
     citeRecStr.startsWith("<div")) { // an error message from JDBC
    idStr="";
    citeRecStr= "<rdf:Description/>"; // empty record
  }
  // with citeRecStr initialized, construct a Transformer to output form
  TransformerFactory tFactory = TransformerFactory.newInstance();
  Transformer transformer = tFactory.newTransformer(new StreamSource(xslUri));
  // set top-level parameters of the stylesheet
  transformer.setParameter("isCommitter",isCommitter?"yes":"");
  transformer.setParameter("idStr",idStr);
  transformer.setParameter("topicNames",appCache.get("topicNames").toString());
  transformer.transform
  (
    new StreamSource(new StringReader(citeRecStr)),
    new StreamResult(out)
  );
%><!-- output links to logout and rdbCtl.jsp -->
<a href="logout.jsp" target="_top">Logout</a> or
<a href="rdbCtl.jsp">return</a>.<br>
</body></html>
```

At this point, the action switches to uiCitation.xsl.

XSLT for Citation Input Form

This stylesheet generates a form for creating or editing a citation entry. Its input is either an empty entry, <rdf:Description/>, or an existing entry. (See Listing 7-18.)

Listing 7-18. Example of Citation

```
<rdf:Description>
<dc:identifier>C0</dc:identifier>
<dc:identifier>ISBN 0333776267</dc:identifier>
<dc:creator>Duncan Langford</dc:creator>
<dc:title>Internet Ethics</dc:title>
<dc:date>2000-06-06</dc:date>
<dc:publisher>ILRT, University of Bristol</dc:publisher>
<dc:format>Book</dc:format>
```

```
<dc:subject>internet; www; ethics; technology; </dc:subject>
<dc:description>Covers "moral varieties of Internet mischief"
and many other topics; contributing authors from 11 countries.
<a href="http://www.amazon.com/exec/obidos/ASIN/0312232799">See
Amazon.com listing</a>.</dc:description>
</rdf:Description>
```

In this template, items other than subject and description can be repeated, and everything is optional except the first `dc:identifier` field. Figure 7-4 shows the output of the stylesheet as displayed if the user is a committer and has entered a valid record ID.

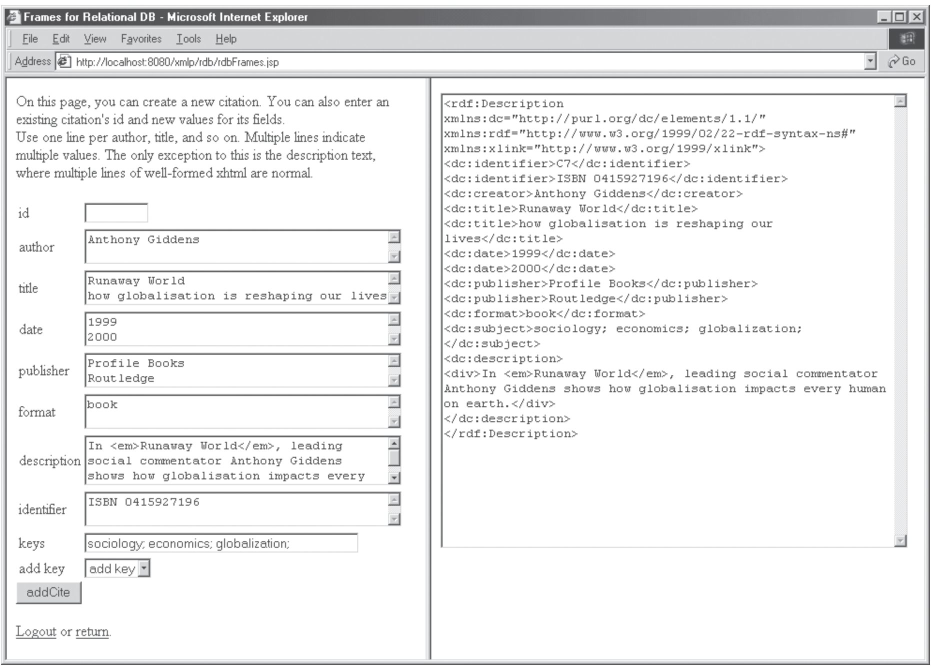

Figure 7-4. Entry form for citation

The XSLT, as we just saw, has top-level parameters that provide the following information: the user's status (committer or not), the ID string of the record to work on, and the list of topics. If the user is not a committer, the last two parameters are empty strings.

In outline, the XSLT consists of three parts:

- the root and top-level elements, before the first template
- the root template that outputs most of the HTML page, including text areas for entering multiple values of a Dublin Core element
- a named template for outputting a SELECT element for subject options

All XSLT details in this stylesheet should be familiar from the two XSLT chapters. In Listing 7-19, the only detail that merits a reminder is the exclude-result-prefixes attribute on the root. Its purpose is to eliminate extraneous namespace nodes within the subtree of the stylesheet rooted at the element bearing the exclude-result-prefixes attribute. (In our case, this is the entire stylesheet element tree.)

Listing 7-19. The Root and Top-Level Elements of uiCitation.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    exclude-result-prefixes="dc rdf"
>
<xsl:output   method="html"
  indent="yes" encoding="UTF-8"
  omit-xml-declaration="yes"
/>
<xsl:param name="topicNames" select="''"/>
<xsl:param name="idStr" select="''"/>
<xsl:param name="isCommitter" select="''"/>
```

The Root Template

Listing 7-20, 7-21, and 7-22 contain the root template. It is fairly long but repetitive: most of its code outputs text areas to enter citation components. Before we get to this part, we construct a variable whose value controls whether or not we retrieve data from the database to populate the form. Two conditions need to be checked: is the user a committer and is there a valid ID value? (See Listing 7-20.)

Listing 7-20. The `okIdStr` *Variable*

```
<xsl:template match="/">
<xsl:variable name=" ">
  <xsl:choose>
    <xsl:when test="$isCommitter='yes' and starts-with($idStr,'C')">
       <xsl:value-of select="$idStr"/>
    </xsl:when>
    <xsl:otherwise>
       <xsl:value-of select="''"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

This piece of code is followed by instructions to the user (within a `<div>` element) that say “Use one line per author, title, and so on. Multiple lines indicate multiple values. The only exception to this is the description text, in which multiple lines of well-formed XHTML are normal.” This is followed by a form that contains a two-column table. The first cell of each row is a label describing the contents of the second cell; the second cell is an input element or a text area. Listing 7-21 shows the first two rows of the table. The first row is, in fact, conditionally present only if the user is a committer. The second row shows a text area into which the author or authors are entered, separated by the newline character.

Listing 7-21. The Form, the Table, and the First Two Rows

```
<form name="addCite" action="upAddCite.jsp" target="dataFrame">
<table>
<xsl:if test="$isCommitter">
<tr>
  <td>id</td>
  <td><input type="text" name="idStr"   value="{$okIdStr}" size="8"/></td>
</tr>
</xsl:if>
<tr>
  <td>author</td>
  <td><textarea name="author" rows="2" cols="40">
    <xsl:for-each select="rdf:Description/dc:creator">
       <xsl:value-of select="concat(.,'&#10;')" />
    </xsl:for-each>
    </textarea></td>
</tr>
```

There are similar text area rows for title, date, publisher, and format, but the rows for description, identifier, and subject keywords are different. In the case of description, we want to copy the entire content of the element into the text area, without breaking it into lines:

```
<tr>
  <td>description</td>
  <td><textarea name="description" rows="3" cols="40">
       <xsl:copy-of select="rdf:Description/dc:description/*/node()" />
    </textarea></td>
</tr>
```

In the case of identifiers, we want to exclude the first identifier that is the system-generated ID and output only the remaining ones. We use the predicate `position()>1`, with the “>” character appropriately encoded:

```
<tr>
  <td>identifier</td>
  <td><textarea name="identifier" rows="2" cols="40">
    <xsl:for-each select="rdf:Description/dc:identifier[position()&gt;1]">
       <xsl:value-of select="concat(.,'&#10;')" />
    </xsl:for-each>
 </textarea></td>
</tr>
```

Finally, for the keys, we want to output a drop-down selection list, which requires slightly more-involved code but serves as a good exercise on recursion. Because we don’t know how many subject keywords there are, we create a named template that outputs a single option within the select element, and recursively call it until the list of “topicNames” is empty:

```
<tr>
  <td>keys</td>
  <td>
  <input type="text" name="keys" value="{rdf:Description/dc:subject}" size="40"/>
</td>
</tr>
<tr>
  <td>add key</td>
  <td><select name="addKey" size="1"
      onchange="with(this.form) keys.value+=addKey.value+'; '">
  <option selected="1" value="">add key</option>
<xsl:call-template name="subjectOptions">
```

```
  <xsl:with-param name="topicOptions" select="$topicNames"/>
</xsl:call-template>
 </select>
</td>
</tr></table>
```

The rest of the root template outputs the Submit button and closes off the form and the template.

The Recursive subjectOptions Template

Finally, the subjectOptions template does a straight linear recursion on the semicolon-separated list of subject keywords:

```
<xsl:template name="subjectOptions">
  <xsl:param name="topicOptions" select="''"/>
  <xsl:if test="contains($topicOptions,'; ')">
    <xsl:variable name="opt" select="substring-before($topicOptions,'; ')"/>
    <option value="{$opt}"><xsl:value-of select="$opt"/></option>
    <xsl:call-template name="subjectOptions">
      <xsl:with-param name="topicOptions"
           select="substring-after($topicOptions,'; ')"/>
    </xsl:call-template>
 </xsl:if>
</xsl:template>
```

This concludes uiCitation.xsl and the entire user interface sequence for creating or editing a citation. The input from the form generated by uiCitation.xsl is processed by upAddCite.jsp.

Creating or Editing a Citation: upAddCite.jsp

In outline, this JSP proceeds as follows:

1. Perform the usual preliminaries: inputs, two beans, the getConn.jsp include file, login check, session objects (DOM parser, null Document, and Transformer).

2. Retrieve information from Request (components of citation).

3. Compute ID, determine if this is a new or edited citation.

4. Build a DOM tree for the citation. Fields containing multiple values (one per line) are parsed using a StringTokenizer.

5. Serialize DOM object to XML string; add record to the database using addCitation() method; output the XML string to text area in the data frame.

6. Define `addCitation()` method.

We will skip the preliminaries because you have seen them all several times. The rest of the code is presented in separate listings corresponding to items in this outline, starting with "Retrieve Request" in Listing 7-22. We cover `addCitation()` in a separate subsection because some parts of the code are a bit involved.

Listing 7-22. Retrieve Request Information, Trimming Whitespace

```
String idStr=request.getParameter("idStr"); // citation ID, or ""
String identifierStr=request.getParameter("identifier").trim();
String authorStr=request.getParameter("author").trim();
String titleStr=request.getParameter("title").trim();
String dateStr=request.getParameter("date").trim();
String publisherStr=request.getParameter("publisher").trim();
String formatStr=request.getParameter("format").trim();
String keysStr=request.getParameter("keys"); // ends with "; "
String descriptionStr=request.getParameter("description").trim();
```

Before we build a new DOM for this citation, we compute its ID and determine whether this is a new record (in which case the ID is the current `idCount` and the `idCount` needs to be incremented) or an edit of an existing record. (See Listing 7-23.)

Listing 7-23. Compute ID; Determine If This Is a New or Edited Record

```
int identInt=-1;
boolean isNewRecord=false;
try{
  if(null==idStr){} // new record; id is -1
  else if(idStr.startsWith("C")) // compute id as integer
    identInt=Integer.parseInt(idStr.substring(1));
}catch(Exception ex){}
// if this is a new record, increment idCount in appCache
// within a synchronized block
synchronized(appCache){
  int idCount=( (Integer)appCache.get("idCount") ).intValue();
```

```
    if(identInt<0 || identInt >= idCount){ // new record
      isNewRecord=true;
      appCache.put("idCount",new Integer(1+idCount));
      identInt=idCount;
    }
  }
  idStr="C"+Integer.toString(identInt);
```

We are ready to start building a DOM tree. This is the longest part by far, but the code (see Listing 7-24) is repetitive, and we will make a few cuts. For most fields, we break the submitted string into individual items using a StringTokenizer and call `appendChild()`; we will show only one of them. The ID, subject, and description fields are treated differently. Before we do anything else, we declare namespace URI strings and use them in constructing the `rdf:Description` element.

Listing 7-24. Build DOM tree, with Namespaces

```
  String dcURI = "http://purl.org/dc/elements/1.1/";
  String rdfURI =  "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
// construct the citation element with namespace declarations
  Element cite=doc.createElementNS(rdfURI,"rdf:Description");
  cite.setAttribute("xmlns:rdf",rdfURI);
  cite.setAttribute("xmlns:dc",dcURI);
// construct citation element's components, beginning with ident
  Element identCite=doc.createElementNS(dcURI,"dc:identifier");
  identCite.appendChild(doc.createTextNode(idStr));
  cite.appendChild(identCite);
  // create a Tokenizer for multivalue fields
  StringTokenizer st=null; // initialized for each multivalue field
  // an example of a multivalue field: dc:identifier
  st=new StringTokenizer(identifierStr,"\n\r");
  while (st.hasMoreTokens()){
    Element identifier=doc.createElementNS(dcURI,"dc:identifier");
    identifier.appendChild(doc.createTextNode(st.nextToken()));
    cite.appendChild(identifier);
  }
// skip similar code for author, title, date, publisher,format
// for subject, simply create a text node child out of keyStr
  Element subject=doc.createElementNS(dcURI,"dc:subject");
  subject.appendChild(doc.createTextNode(keysStr));
  cite.appendChild(subject);
// for description, we embed the XHTML string into a div element,
// parse into a DOM node and attach using DOM's importNode() method
```

```
  Element description=doc.createElementNS(dcURI,"dc:description");
  if(descriptionStr!=null && descriptionStr.length()>0) {
    try{
      descriptionStr="<div>"+descriptionStr+"</div>";
      Document desc=db.parse(new InputSource(new StringReader(descriptionStr)));
      description.appendChild(doc.importNode(desc.getDocumentElement(),true));
    }catch(Exception ex){}
  }
  cite.appendChild(description);  cite.appendChild(description);
  }
```

With the DOM object constructed, we can serialize it using the default transformer and use the resulting XML to append a record to the database. We will also display serialized XML in a text area.

Because the default transformer cannot output directly to a String object, we output to a stream (StringWriter) and convert to a string. Otherwise, the code is straightforward and familiar, as shown in Listing 7-25.

Listing 7-25. Serialize DOM, Add to Database, and Output to Text Area

```
  // serialize DOM object to string, via StringWriter
  StringWriter sw=new StringWriter();
  trans.transform(new DOMSource(cite),new StreamResult(sw));
  String xmlValue=sw.toString();
  // add citation to database
  addCitation(isNewRecord,identInt,identifierStr,authorStr,titleStr,
              dateStr,keysStr,xmlValue,unm,sessCache,appCache);
  // output serialized citation to text area in data frame
%>
<html><head><title>Add Citation</title></head><body>
<textarea rows="30" cols="60">
<%
  trans.transform(new DOMSource(cite),new StreamResult(out));
%>
  </textarea></body></html>
```

The only remaining piece is the definition of the `addCitation()` method. This is where we use JDBC, and particularly PreparedStatement.

The `addCitation()` *Method*

The `addCitation()` method, shown in Listing 7-26, adds both a citation and a submission record to the database. In either case, we need different SQL

depending on whether we create a new citation or edit an existing one. (In the first case, we need an INSERT statement; in the second case, we need an UPDATE statement.) Altogether, there are four different SQL statements for which we create a PreparedStatement. The corresponding SQL strings are created in the code (some of them built up out of smaller pieces because they are too long to fit on a single line), but we will display two of them separately to make reading the code easier. The INSERT statement for a citation is

```
"INSERT INTO rdbCitations VALUES(?,?,?,?,?,?,?)"
```

and the UPDATE statement for citation, broken into two lines, is

```
"UPDATE rdbCitations SET ident=?,identifier=?,creator=?,title=?,
crDate=?, subject=?,xmlValue=? WHERE ident="+identInt
```

In the second SQL string, the value of `identInt` is computed by the JSP and appended to the string.

For each of the four PreparedStatements, we must set values of statement parameters. We use `setString()` and `setInt()` for the purpose, with two exceptions. One of them is the `xmlValue` field in the `rdbCitations` table, which is a string that can be longer than 255 characters. We have to use `setObject()`, with three arguments: field number or name, value to be converted to object, and the “real” data type of the value (`java.sql.Types.LONGVARCHAR`). When this value is retrieved, the stored object is converted to its intended data type and then to the corresponding Java type.

In the `rdbSubmissions` table, the fields having to do with commitment (date and committer identity) have to be set to NULL because commitment is performed as a separate action. In this case, `setNull()` is used, with two arguments: field number or name, and the field’s data type.

Listing 7-26. Definition of `addCitation()`

```
<%! // addCitation() method: XML string to database
public void addCitation(
                          boolean isNewRecord,
                          int identInt,
                          String identifierStr,
                          String authorStr,
                          String titleStr,
                          String dateStr,
                          String keysStr,
                          String xmlValue,
                          String submitter,
                          Hashtable sessCache,
```

```
                         Hashtable appCache
                        ) throws Exception{
Connection con=null; PreparedStatement pStmt=null;
try{
  con=getConnection(appCache);
// create PreparedStatement for Citation, new or edited
  if(isNewRecord)
  pStmt=con.prepareStatement("INSERT INTO rdbCitations VALUES(?,?,?,?,?,?,?)");
  else pStmt=con.prepareStatement(
  "UPDATE rdbCitations SET "
   +" ident=?,identifier=?,creator=?,title=?,crDate=?,subject=?,xmlValue=?"
   +" WHERE ident="+identInt);
// fill in parameters of PreparedStatement with data from Request and elsewhere
  pStmt.setInt(1,identInt);
  pStmt.setString(2,identifierStr);
  pStmt.setString(3,authorStr);
  pStmt.setString(4,titleStr);
  pStmt.setString(5,dateStr);
  pStmt.setString(6,keysStr);
// to setString whose len is > 255, use setObject(fieldNumber, obj, JDBC type)
  pStmt.setObject(7,xmlValue,java.sql.Types.LONGVARCHAR);
  pStmt.executeUpdate();
  pStmt.close(); // done with Citation; Submission record next
// create PreparedStatement for Submission, new or edited
  if(isNewRecord)
   pStmt=con.prepareStatement("INSERT INTO rdbSubmissions VALUES(?,?,?,?,?)");
  else pStmt=con.prepareStatement(
   "UPDATE rdbSubmissions SET "
   +" ident=?,author=?,subDate=?,committedBy=?,comDate=?"
   +" WHERE ident="+identInt);
// fill in parameters of PreparedStatement with data from Request and elsewhere
  pStmt.setInt(1,identInt);
  pStmt.setString(2,submitter);
  DateFormat dateFormat=(DateFormat)sessCache.get("dateFormat");
  String rightNowStr=dateFormat.format(new Date());
  pStmt.setString(3,rightNowStr);
// set next two fields to NULL; specify data type
  pStmt.setNull(4,java.sql.Types.VARCHAR);
  pStmt.setNull(5,java.sql.Types.TIMESTAMP);
  pStmt.executeUpdate();  }catch(Exception ex){}
  finally{ // the ususal clean up
   if(pStmt!=null) try{pStmt.close();pStmt=null;}catch(Exception ex){}
```

```
    if(con!=null) try{con.close();con=null;}catch(Exception ex){}
  }
} %>
```

This concludes the code of upAddCite.jsp and the entire update section. Next, we look at the SELECT queries and refset actions. Our presentation here will be less complete because many details are already familiar. The main point of interest is in retrieving XML data from a relational database using a combination of an SQL query and an XPath condition.

Query Implementations 2: Refset Actions

To remind you of the overall picture, we repeat the diagram of Figure 7-1 in Figure 7-5 to show refset operations.

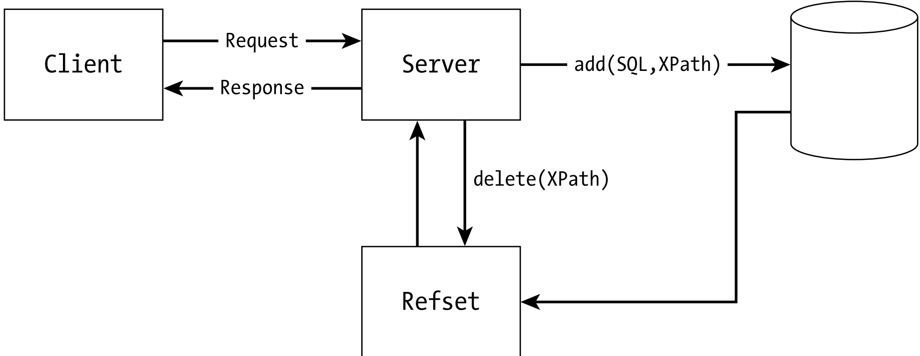

Figure 7-5. Select actions: client, server, database, and refset

Supported refset operations are add, delete, clear, and display. If the action is “add,” then the database is searched for matching records, and they are added to the refset. (The search condition consists of two parts: an SQL query and an XPath expression. The database is searched using SQL, and the returned result set is filtered using XPath.) If the action is “delete,” the current refset is searched for matching records (using XPath only), and they are removed from the refset. The “clear” action empties the refset, and “display” action (the default) shows its contents in a text area.

In terms of code, we will work through rdbRefSetOps.jsp that implements all the actions. In particular, it defines the following methods: `addRefs()`, `delRefs()`, and `writeRefs()` for display. (Because refset is implemented as a hashtable, the “clear” action is implemented by simply calling the hashtable `clear()` method.)

The supporting `getRefIf()` method does the XPath filtering, with the help of code in getRef.jsp.

In outline, rdbRefSetOps.jsp proceeds as follows:

1. Perform the usual preliminaries: imports, beans, includes, the login check.

2. Get information from Request, including the selected action parameter.

3. Depending on the value of the action parameter, call the appropriate function.

4. Display the current refset in a text area in the data frame.

5. Define all the requisite functions.

Skipping the preliminaries, Listing 7-27 shows the first part of the file, up to function definitions.

Listing 7-27. Part 1 of rdbRefSetOps.jsp: Get Request Information, Call the Right Action Method

```
// get information from Request
 String sqlWhere=request.getParameter("sqlWhereClause");
 if(sqlWhere==null) sqlWhere="";
 String xPath=request.getParameter("XPath");
 if(xPath==null) xPath="";
 String refType=request.getParameter("type"); // citation or annotation?
 String action=request.getParameter("action");

// dispatch on action -- call appropriate method
 if("clear".equals(action)) {refSet.clear();}
 else if("show".equals(action)) {/* do nothing */}
 else if("add".equals(action))
   addRefs(sqlWhere,refType,xPath,refSet,sessCache,appCache);
 else delRefs(xPath,refSet,sessCache,appCache);

// display the refset
%><html><head><title>Search result</title></head><body>
The current reference set is as follows:
<form><textarea rows="30" cols="60">
<% writeRefs(refSet,appCache,out);
%></textarea></form></body></html>
```

Definitions of Refset Operations

Definitions of refset operations are quite straightforward except for the supporting methods (`getRefIf()` and `checkXpath()`) that do XPath filtering. In this section, we show the top-level definitions, highlighting the lines that use supporting methods. The key (no pun intended) to their operation is that the same record ID serves as the primary key in the database and the key under which the record is stored in the refset.

In Listing 7-28, we show two shorter methods: `writeRefs()` and `delRefs()`.

Listing 7-28. Part 2 of rdbRefSetOps.jsp: Action Definitions

```
public void writeRefs(Hashtable refSet, Hashtable appCache, Writer out){
// output refset to the out stream
Connection con=null; Statement stmt=null;
try{
  con=getConnection(appCache);
  stmt=con.createStatement();
  out.write("<refList>\n");
  for(java.util.Enumeration e=refSet.keys();e.hasMoreElements() ;){
    String key=e.nextElement().toString();
    out.write(getXmlValue(key,queryStringIdent(key),con,stmt));
  }
   out.write("</refList>\n");
}catch(Exception ex){
 try{out.write("<div class='badError'>getRefs err: "+ex+"</div>");}
 catch(Exception ex2){}
}finally{
   if(stmt!=null)try{stmt.close();}catch(Exception ex){}
   if(con!=null)try{freeConnection(con,appCache);}catch(Exception ex){}
  }
}

public void delRefs(String xPath,Hashtable refSet,
                       Hashtable sessCache,Hashtable appCache)throws Exception{
  Enumeration keys=refSet.keys();
  while(keys.hasMoreElements()){
    String key=keys.nextElement().toString();
// for each key, retrieve XML string from the database, test with XPath
// return non-null if matched
    if(null!=getRefIf(key,xPath,sessCache,appCache))
      refSet.remove(key);
  }
}
```

Listing 7-29 shows `addRefs()`, a long but fairly transparent piece of code. It cleanly falls into two parts: the first part does database access and knows nothing about XML, and the second does XPath filtering on the result set obtained in the first part.

Listing 7-29. rdbRefSetOps.jsp, `addRefs()`

```
public void addRefs(
    String sqlWhere, // SQL where clause
    String refType,   // citation or annotation?
    String xPath,      // XPath condition to check
    Hashtable refSet,Hashtable sessCache,Hashtable appCache)
       throws Exception{
// Part 1: run SQL query, obtain result set
  Connection con=null;
  PreparedStatement pStmt=null;
  ResultSet rSet=null;
  try{
    con=getConnection(appCache);
    String tableName; String prefix;
    if("annotations".equals(refType)){
      tableName="rdbAnnotations";  prefix="A";
    }
    else {
      tableName="rdbCitations"; prefix="C";
    }
    String queryString="SELECT ident,xmlValue FROM "+tableName;
    if(sqlWhere.length() > 0) queryString+=" WHERE "+sqlWhere;
    pStmt=con.prepareStatement(queryString);
    rSet=pStmt.executeQuery();
// Part 2: filter result set by XPath
    while(rSet.next()){
       int ident = rSet.getInt(1); // get first item from next record
       String xmlValue=rSet.getString(2);  // get second item
       String identStr=prefix+ident; // e.g., "A"+254
       if(xPath.length()==0) // no XPath to test; add to refset
        refSet.put(identStr,xPath);
      else {
        DocumentBuilderFactory dbf=
          (DocumentBuilderFactory)sessCache.get("dbf");
        DocumentBuilder db=dbf.newDocumentBuilder();
         Document doc=db.parse(new InputSource(new StringReader(xmlValue)));
        Node node=doc.getDocumentElement();
        if(checkXpath(node,xPath,appCache))
```

```
        refSet.put(identStr,xPath);
      }
    } // end while
  }finally{ // clean up
    if(rSet!=null)try{rSet.close();rSet=null;}catch(Exception ex){}
    if(pStmt!=null)try{pStmt.close();pStmt=null;}catch(Exception ex){}
    if(con!=null)try{freeConnection(con,appCache);}catch(Exception ex){}
  }
}
```

XPath Filtering

We do XPath filtering in an Apache-specific way. There are several other ways to achieve the same result. When the *XQuery 1.0* and *XPath 2.0* recommendations are completed, the task of retrieving XML data from a database using an XPath expression will be further streamlined, and perhaps standardized. For this reason, we encapsulate the filtering predicate into a separate method, `checkXpath()`, and we do not place great pedagogical value on our code that implements it—but it can be easily replaced.

The predicate, as you have just seen in `addRefs()`, takes three arguments: an element to test, an XPath expression to test it against, and the appCache. The appCache is needed to process namespaces: as you remember, the namespaces are stored as a `PrefixResolverDefault` object in appCache. (See `setAppCacheRootElt()` in Listing 7-11.) That object, together with the element and the XPath, is given as an argument to the public static `eval()` method of `org.apache.xpath.XPathAPI`; the output of the method is converted to a Boolean and returned. The code is shown in Listing 7-30.

Listing 7-30. `checkXpath()`

```
public boolean checkXPath(Node node,String xPath,Hashtable appCache){
  try{ // using Xalan's XPathAPI; the alternative is
         // to insert xPath into a stylesheet in a Transformer object
     PrefixResolver resolver=(PrefixResolver)appCache.get("namespaces");
     // evaluate and convert to boolean
     return XPathAPI.eval(node,xPath,resolver).bool();
  }catch(Exception ex){return false;}
}
```

The same predicate is used in `getRefIf()` that returns the DOM tree of the document indicated by a reference, but only if that document checks against an XPath condition. (See Listing 7-31.)

Listing 7-31. `getRefIf()`

```
public Document getRefIf(String ref,String xPath,
                           Hashtable sessCache,Hashtable appCache){
// Return DOM Document indicated by ref,
// but only if "xPath" condition holds.
  try{
    Document doc=getDoc(ref,sessCache,appCache);
    if(checkXPath(doc.getDocumentElement(),xPath,appCache))
      return doc;
  }catch(Exception ex){}
  return null;
}
```

Finally, the unconditional retrieval of a DOM document by its reference is done by `getDoc()` and in a completely unremarkable way: the serialized XML document is retrieved from the database and parsed, as shown in Listing 7-32.

Listing 7-32. `getDoc()`

```
public Document getDoc(String ref,Hashtable sessCache,Hashtable appCache){
// Return DOM Document indicated by ref.
  try{
    String xmlValue=getXmlValue(ref,appCache);
    DocumentBuilderFactory dbf=
      (DocumentBuilderFactory)sessCache.get("dbf");
    DocumentBuilder db=dbf.newDocumentBuilder();
    return db.parse(new InputSource(new StringReader(xmlValue)));
  }catch(Exception ex){return null;}
}
```

This concludes our discussion of refset operations, and the entire chapter.

Conclusion

We covered a lot of ground in this chapter. The exposition has been organized around a larger application that uses a relational database as a repository of persistent XML data. The repository can be queried by a combination of SQL and XPath expressions. The data itself is metadata, a collection of references to online and offline resources and annotations on those references and other annotations.

To cover all the aspects of the application, we have introduced some new material and revisited many previously discussed topics. In particular:

- To represent the data, we introduced XML metadata formalisms, Resource Description Framework (RDF), and Dublin Core Metadata Element Set (DCMES).

- To represent graphs of annotations, we revisited XLink, with modifications: the `href` attributes defining locators are keys for looking up XML data in a relational database.

- We have discussed ways to store XML data in relational databases and illustrated some of them in our database: some of our XML records are stored in the database as a collection of fields and others as XML text to be parsed on retrieval.

- Before showing how the database is queried, we discussed the general principles of database access and how they are implemented in Java. Similar facilities, of course, exist in the ASP/ADO and the .NET frameworks.

- Throughout the application, we used DOM both to build new XML records and to modify existing ones.

We mentioned several times in the chapter that, eventually, a "native XML database" may be a better infrastructure than a relational database for storing XML data. However, native XML databases are still in beta at best, and, in any event, relational database will continue to be in use for the foreseeable future, and the task of integrating them with XML data stores and data flows will remain relevant. Also on the horizon are the *XQuery* and *XPath 2.0* recommendations, which will bring a new degree of standardization to querying XML data using XPath expressions (combined with SQL-like expressions of XQuery). When these specifications are completed, a new edition of this book will definitely be in order. Even then, the ideas and techniques introduced in this chapter will remain useful for XML programmers.

CHAPTER 8

RELAX NG and XML Schema

THIS CHAPTER PRESENTS two recent alternatives to DTDs: RELAX NG and XML Schema. We will start with RELAX NG, which is a smaller framework that can be easily covered in its entirety. *XML Schema* is a much larger specification that would take a few chapters to do any kind of justice to it. (The specification itself is more than 350 pages long.) Fortunately, it consists of two parts, and Part 2 ("Datatypes") is completely independent of Part 1 ("Structures"). Part 2 is easy to summarize, it can be used independently of Part 1, and both RELAX NG and SOAP make use of it. XML Part 1 is a few notches higher on the scale of complexity from Part 2, or from any other XML specification. However, it's possible to ignore much of it, and that's what we are going to do, concentrating on only those areas that are needed for understanding SOAP and the Web services technology.

As we have mentioned, *XML Schema* is a W3C recommendation. The RELAX NG specification was developed under the auspices of an OASIS technical committee and released December 3, 2001. It has since been submitted to the ISO for approval. As far as we know, this is the first time in the brief history of XML that a major specification is moving towards a standard outside of W3C.

Both RELAX NG and XML Schema have the following features:

- XML syntax
- support for namespaces
- modularity and ease of reuse
- support for a rich system of data types comparable to those of major programming languages

In both specifications, element content models are "first-class objects": tree-structured entities that can be named and referenced by name (and therefore reused in multiple contexts). This is a major advance over DTD parameter entities that are simply character strings.

For all these similarities, the two languages still have major differences between them. *XML Schema Part 1* is a huge monolithic whole. It contains many features that have little to do with validation (such as type inheritance and identity constraints). It mandates massive additions to the document's infoset as a byproduct of validation. The data-typing vocabulary of *XML Schema Part 2* is hardwired into Part 1. By contrast, RELAX NG emphasizes modularity and extensibility. Its main task is validation, and it takes care not to introduce anything in a RELAX NG grammar that would make additions to the document's infoset. For instance, it provides no means for specifying attribute defaults. If such additions are desired for backward compatibility with DTDs, a specialized mechanism can be invoked to achieve such compatibility. Similarly, RELAX NG has a clean interface for connecting to a data type library that may or may not be *XML Schema Part 2*. If all you need to do is validate and check simple data types, RELAX NG is a better tool. It is also by far the best tool for validating XHTML-derived languages, within the framework of XHTML modularization described in Chapter 3. RELAX NG makes it possible and practicable for ordinary mortals to use the modularization framework. In summary, RELAX NG does fewer things but it does them better, and it provides hooks for additional functionality (such as data typing) that can be performed before, after, or instead of validation.

In outline, the chapter will proceed as follows:

- RELAX NG introduction and frameworks for testing
- RELAX NG overview, with subdivisions
- modularization and reuse with RELAX NG
- XHTML modularization with RELAX NG
- data typing within RELAX NG using XML Schema data types
- overview of the *XML Schema Part 1* recommendation

RELAX NG History and Current Condition

RELAX NG resulted from the merger of two earlier projects: Murata Makoto's RELAX (REgular LAnguage description for XML) and James Clark's TREX (Tree Regular Expressions for XML). The two projects had a good deal in common: both aimed to produce a better DTD, with XML syntax, support for namespaces, and more expressive power. Both relied on *XML Schema Part 2* for a vocabulary of data types. After the merger, the creators of TREX and RELAX became the co-editors of

RELAX NG (Next Generation), within the framework of an Oasis technical committee (`http://www.oasis-open.org/committees/relax-ng/`). In December 2001, the committee published version 1.0 of the specification, together with two supplemental documents: a tutorial and a DTD compatibility specification.

Other Materials and Tools

In addition to the official OASIS site, a major source of RELAX NG materials is James Clark's Web site, `www.thaiopensource.com`. It contains, among other things,

- two versions of a RELAX NG validator Jing (presented shortly).
- an implementation of the XHTML modularization framework and XHTML Basic.
- RELAX NG grammars for RELAX NG itself and for XSLT.
- a test suite for RELAX NG.
- a remarkable program called DTDinst that converts a DTD (with parameter entities and all) into an equivalent XML document. Two XML output formats are provided, one of them RELAX NG.
- a paper entitled "The Design of RELAX NG" that provides motivations for major design decisions. Although theoretical in places, the paper is useful for RELAX NG practitioners as well.

Several RELAX NG tools can be found at Sun's Developer Connection site, `http://www.sun.com/software/xml/developers/`. The most important of them is the Multi-Schema Validator (MSV) that can validate XML documents against several kinds of XML schemas: RELAX NG, RELAX, TREX, DTD, and a subset of *XML Schema Part 1*.

An interesting tool related to RELAX NG is RelaxNGCC (RELAX NG Compiler Compiler, `http://homepage2.nifty.com/okajima/relaxngcc/index_en.htm`). A "compiler-compiler" is a computer program whose input is a grammar in which every grammar rule is associated with a certain action, described by a piece of code (such as a function). The program uses such an input to generate a "compiler" for the grammar that, in addition to parsing, performs actions associated with grammar rules. Because, in a context-free grammar, a grammar rule corresponds to a subtree of the parse tree, the compiler-compiler generates a program that performs specified operations on subtrees of the parse tree. The best-known compiler-compiler is YACC (Yet Another Compiler Compiler), which

is part of the Unix operating system. Many compiler-compilers have been written for BNF grammars (introduced in Chapter 2); RelaxNGCC provides similar functionality for RELAX NG grammars.

With background in place, we proceed to a simple example and a framework for testing.

A Simple Example

Recall our very first DTD for a question-answer exchange (Listing 1-5 in Chapter 1):

```
<!ELEMENT exchange (q, a)>
<!ELEMENT q (#PCDATA)>
<!ELEMENT a (#PCDATA)>
<!ATTLIST exchange tone (friendly|polite| cold|rude) "friendly">
```

In RELAX NG, this comes out as `rng/exchange.rng`.

Listing 8-1. The First RELAX NG Grammar

```
<element name="exchange xmlns="http://relaxng.org/ns/structure/1.0">
  <element name="q"><text/></element>
  <element name="a"><text/></element>
  <attribute name="tone">
    <choice>
      <value>friendly</value>
      <value>polite</value>
      <value>cold</value>
      <value>rude</value>
    </choice>
  </attribute>
</element>
```

Two differences from the DTD are immediately noticeable:

- A RELAX NG grammar is an XML document in `"http://relaxng.org/ns/structure/1.0"` namespace.
- There is no way to provide a default value for an attribute. In general, there is no way, within RELAX NG proper, to put material in the grammar that will augment the document's infoset. However, you can do that in grammar annotations, as explained later in the chapter.

Validating Against RELAX NG Grammars

Several validation tools are available, including James Clark's Jing and Sun's MSV (`http://www.sun.com/software/xml/developers/multischema/`). We use Jing, which comes in two versions: a jing.jar file and a Windows executable, jing.exe. The Windows executable is used from the command line wheareas the jar executable can be used either from the command line or within a Web application. We will present both kinds of usage.

Using a Web Application

To connect to our Web application for RELAX NG, refer to `http://localhost:8080/xmlp/rng/validate.htm`. You will see a page with two frames. The frame on the left has a form with two input boxes: one for an XML file to validate and the other for a RELAX NG grammar to validate it against. Submitting the form runs the validator; if the text area on the right shows no output, validation has been successful. Otherwise, an error message is displayed. Figure 8-1 shows the page with the output from termDefBad.xml and termDef.rng.

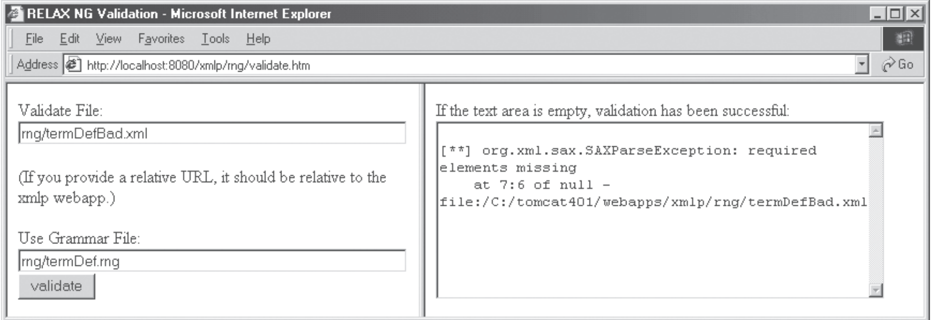

Figure 8-1. RELAX NG validation page

The form input, when submitted, goes to validaterng.jsp. The main fact to keep in mind when reading that JSP is that Jing works on top of a SAX2 XML parser. Because we want to capture its error messages and display them in a Web page, we have to go inside Jing code and obtain our own instance of the parser (XML reader) and the error handler.

In outline, validaterng.jsp proceeds as follows:

1. Import the libraries.
2. Declare methods for getting an XML reader and an error handler.
3. Process the Request object.
4. Set up output streams.
5. Parse, validate, and display errors (if any) in the text area.

Skipping the imports, we proceed directly to declarations. The structure of Jing's code is such that, to obtain a parser instance, we have to implement an `XMLReaderCreator` interface and give the implementing class a `createXMLReader()` method, as shown in Listing 8-2.

Listing 8-2. Obtaining an XML Reader

```
public class XMLReaderCreatorImpl implements XMLReaderCreator {
  public XMLReaderCreatorImpl() {} // default constructor
  public XMLReader createXMLReader() throws SAXException {
    XMLReader xr;
    try{
      xr=SAXParserFactory.newInstance().newSAXParser().getXMLReader();
    }catch(Exception ex){throw new SAXException(ex);}
    // set the parser's features, as explained in chapter 4 :
    // namespace-aware, do not preserve prefixes
    xr.setFeature("http://xml.org/sax/features/namespaces", true);
    xr.setFeature("http://xml.org/sax/features/namespace-prefixes", false);
    try { // disable DTD validation
      xr.setFeature("http://xml.org/sax/features/validation", false);
    }catch (Exception e) {}
    return xr;
  }
}
```

The error handler is created by a plain public method (not within a class). The method creates a modified default error handler and returns it. The Java code is a little tricky in its use of an anonymous inner class, but is otherwise quite clear. (See Listing 8-3.)

Listing 8-3. Obtaining an Error Handler

```
public ErrorHandler createErrorHandler(){
  DefaultHandler dh = new DefaultHandler(){
// dh is an object of an anonymous inner class
// that is derived from DefaultHandler
    void printErr(SAXParseException e){
      System.err.println("[**] "+e+"\n    at "+e.getLineNumber()+":"
          +e.getColumnNumber()+" of "+e.getPublicId()+" - "+e.getSystemId() );
    }
    public void warning(SAXParseException e)
      throws SAXException{printErr(e);}
    public void error(SAXParseException e)
      throws SAXException{ printErr(e);}
    public void fatalError(SAXParseException e)
      throws SAXException{ printErr(e);}
  };
  return dh;
}
```

With declarations in place, we can proceed to the code that actually does the work. First, we process the Request object and set up output streams. Remember that, in the Tomcat system, System.out and System.err are directed to the Tomcat console window; we want to direct error output to the text area in the Web page. Once the output streams are set up, we parse and validate.

Listing 8-4. Process the Request Object, Parse, and Validate.

```
// process Request object
String rng=request.getParameter("rng");
  String uri=request.getParameter("uri");
  if(uri.indexOf(":") < 0) // local file, not a URL
    uri = new File(application.getRealPath(uri)).toURL().toString();
  if(rng.indexOf(":") < 0) // local file, not a URL
    rng = new File(application.getRealPath(rng)).toURL().toString();
// set up output streams
  java.io.PrintStream sysOut = System.out;
  java.io.PrintStream sysErr = System.err;
  java.io.ByteArrayOutputStream baos = new java.io.ByteArrayOutputStream();
  final java.io.PrintStream ps = new java.io.PrintStream(baos);
// parse, validate and output error messages, if any:
<html><body>
If the text area is empty, validation has been successful:
<textarea cols="50" rows="10">
<% try{
```

```
    System.setOut(ps);
    System.setErr(ps);
    ValidationEngine engine= new ValidationEngine(
        new XMLReaderCreatorImpl(),createErrorHandler(),true);
    if(!engine.loadSchema(new InputSource(rng))) { // parse grammar
%> sorry, can't load pattern from <%= rng %>; failed <%
}else engine.validate(new InputSource(uri)); // parse and validate XML
}catch(Throwable ex){ex.printStackTrace(new java.io.PrintWriter(out,true));}
  finally{System.setOut(sysOut);System.setErr(sysErr);}
%>
<%= baos.toString() %>
</textarea></body></html>
```

Using the Command Line

From the command line, you can use either jing.jar or (on Windows) jing.exe. To use jing.exe, simply make sure that it is on the path. The usage is

```
jing grammar.rng doc1 doc2...
```

Running jing.exe on exchange.rng and a "bad" exchange that has `q` and `a` in the wrong order (exchangeBad.xml) produces this output (with long lines broken in two and whitespace rearranged for readability):

```
C:\tomcat401\webapps\xmlp\rng>jing exchange.rng exchangeBad.xml
Error at URL "file:/C:/tomcat401/webapps/xmlp/rng/exchangeBad.xml",
line number 3, column number 3: element "a" not allowed at this point
Error at URL "file:/C:/tomcat401/webapps/xmlp/rng/exchangeBad.xml",
line number 5, column number 1: unfinished element
Elapsed time 290 milliseconds
```

To run the Java validator, the jing.jar and a JAXP-compatible SAX2 parser (such as Xerces or Crimson) must be on the classpath. The classpath can be set on the command line with the `-cp` option. The class that runs the validator is `com.thaiopensource.relaxng.util.Driver`. The arguments are the same as with the Windows executable:

```
java -cp c:\cp\jing.jar;c:\cp\xerces.jar com.thaiopensource.relaxng.util.Driver
        grammar.rng doc1 doc2...
```

We provide, in the xmlp/rng directory, two BAT files: one to run Jing with Xerces (jingx.bat) and the other to run Jing with Crimson (jingcrimson.bat). Using jingx.bat, we can run the same test as before like this:

```
C:\tomcat401\webapps\xmlp\rng>jingx exchange.rng exchangeBad.xml
```

This gets expanded and produces nearly identical output:

```
C:\tomcat401\webapps\xmlp\rng>java -cp c:\tomcat401\common\lib\jing.jar;c:\tomca
t401\common\lib\xerces.jar -Dcom.thaiopensource.relaxng.util.RegexEngine=com.tha
iopensource.relaxng.util.XercesRegexEngine com.thaiopensource.relaxng.util.Drive
r exchange.rng exchangeBad.xml
Error at URL "file:/C:/tomcat401/webapps/xmlp/rng/exchangeBad.xml", line number
3, column number 6: element "a" not allowed at this point
Error at URL "file:/C:/tomcat401/webapps/xmlp/rng/exchangeBad.xml", line number
5, column number 13: unfinished element
Elapsed time 651 milliseconds
```

Using the smaller Crimson parser takes a little less time. However, only Xerces supports XML Schema data types.

RELAX NG Overview

There is a peculiar difficulty in writing about RELAX NG: The tutorial (`http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html`) provided by the editors of the specification is so good that it's difficult to add anything to it. We provide an overview that is less comprehensive but that concentrates on features that are likely to be most commonly used.

RELAX NG features can be conveniently grouped around several large themes. (In the bulleted list, the headers in parentheses indicate the corresponding sections of the tutorial.)

- XML syntax
- element definitions and named content models (Named Patterns)
- uniform treatment of elements and attributes (Choice, Attributes, Enumerations, Lists, Interleaving)
- namespace support, both in the grammar itself and in document instances (Namespaces, Annotations)

- name classes (Name Classes)
- modularity and code reuse (Modularity, Nested Grammars)
- a system of data types (Datatyping)

The prefinal version of the specification and tutorial also had material on keys and key references, a generalization of the DTD's ID/IDREF functionality. This material has been removed from the specification (and therefore the tutorial). The reason seems to be that the theoretical foundations of identity and reference in XML are still poorly understood, and therefore the RELAX NG committee felt it was premature to bring this functionality into RELAX NG, which is otherwise based on the well-understood theory of tree automata.

XML Syntax

The language of DTD has several reserved words and symbols that are expressed by XML elements in RELAX NG. Table 8-1 shows a sample of correspondences. As first indications of how elements and attributes are treated uniformly in RELAX NG, note that both CDATA and PCDATA come out as `<text/>`, and the "items" in the table can be replaced by either elements or attributes. (A more precise statement follows soon.)

Table 8-1. A Sample of DTD and RELAX NG Correspondences

MEANING	DTD	RELAX NG
parsed text	PCDATA	<text/>
unparsed text	CDATA	<text/>
empty element content	EMPTY	<empty/>
sequence	item1,item2	item1 item2
choice	item1|item2	<choice>item1 item2</choice>
optional	item?	<optional> item </optional>
zero or more	item*	<zeroOrMore> item </zeroOrMore>
one or more	item+	<oneOrMore> item </oneOrMore>
group	(item1 item2)	<group>item1 item2</group>

Here is an example of a DTD fragment and the corresponding RELAX NG pattern:

```
<!ELEMENT elt ((a,b)|c,d)>
<!ELEMENT a (#PCDATA)>
<!-- same definition for b, c, d; Now RELAX NG -->
<element name="elt">
  <choice>
    <group>
      <element name="a"><text/><element>
      <element name="b"><text/><element>
    </group>
    <element name="c"><text/><element>
  </choice>
  <element name="d"><text/><element>
</element>
```

RELAX NG operates as a pattern-matching language: the grammar specifies patterns that instance documents must match.

Patterns and Grammars

A RELAX NG pattern is an XML element of any of the types shown in Table 8-2.

Table 8-2. RELAX NG Patterns

grammar	The root element, usually
element, attribute	For defining elements and attributes
group, choice	For defining content models; correspond to () and |
optional, zeroOrMore, oneOrMore	For defining content models; correspond to ?, *, and +
list	For listing enumerated values, in both attribute and element definitions
interleave	Sequence in any order, used in both attribute and element definitions
mixed, empty, text	For mixed and empty content models
ref, parentRef, externalRef	References to names defined elsewhere using the `<define>` element
value, data	For typed data
notAllowed	Used in merging grammars; advanced usage

A RELAX NG "document type definition" consists of a single pattern. The pattern can be as simple as a single element definition, as in Listing 8-1. Usually, the root element of a RELAX NG grammar is `grammar`. Within a `grammar` element, a pattern or a sequence of patterns can be named using `define` elements and referenced by name in other patterns using `ref` elements. For instance, an element's content model can be named and reused.

We will illustrate the basic usage with a UsAddress.rng grammar for U.S. postal addresses. The first version (Listing 8-5) is intentionally simplistic.

Listing 8-5. RELAX NG Grammar for U.S. Postal Address

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
<start>
  <ref name="UsAddress"></ref>
</start>
<define name="UsAddress">
  <element name="us-address">
    <ref name="UsAddressContentModel"/>
  </element>
</define>
<define name="UsAddressContentModel">
  <element name="street"><text /></element>
  <element name="city"><text /></element>
  <element name="state"><text /></element>
  <element name="zip"><text /></element>
</define>
</grammar>
```

A `grammar` usually contains a single `start` element to indicate the root element of document instances. (If your grammar is intended for inclusion in other grammars rather than as a top-level grammar, the `start` element can be omitted. See the following text on inclusion and reuse.) The `start` element is followed by any number of `include` and `define` elements. The order of elements, including the `start` element, is not important, but it makes sense to place the `start` element at the top of grammar definition.

Tokens and Enumerations

The grammar of Listing 8-5 does not express many obvious constraints. The content of `zip` is not arbitrary text but a single token that does not contain whitespace. We can improve by saying

```
<element name="zip"><data type="token" /></element>
```

To express the more precise constraint that `zip` consists of five or nine digits, we'll need to import a data-typing library. (See the section on data types later in this chapter.) Without an external data-typing library, RELAX NG has only two built-in data types: `token` and `string`. The difference between them is the same as the difference between NMTOKEN and CDATA in XML: `token` conforms to the syntax of the QName production and does not contain whitespace.

The content of the `state` element is constrained to an enumeration of 51 possible tokens. To express this, we could say

```
 <element name="state">
  <choice>
      <value>AK</value>
      <value>AL</value>
      <value>AR</value>
      <!-- 47 more states and the District of Columbia -->
  </choice>
</element>
```

In DTDs, only attributes could have enumerated values. In RELAX NG, elements and attributes are treated as uniformly as possible: they have the same pattern syntax and nearly identical sets of constraints.

Uniform Treatment of Elements and Attributes

Within an element's definition, both definitions of children elements and definitions of attributes are children of the parent/owner element's definition:

```
<element name="parent-owner">
  <element name="child1"><text/></element>
  <element name="child2"><empty/></element>
  <attribute name="attr1">
  <choice><value>a</value><value>b</value></choice>
  </attribute>
  <attribute name="attr2"></attribute>
</element>
```

The uniform syntax does not mean that the definition of validity has changed: as with DTDs, the order of element declarations determines the order of children in an instance document, but the order of attributes is unconstrained. This means that attribute definitions may appear in any order. They can also be intermixed, in any order, with children elements' definitions. The following definition is equivalent to the previous one:

```
<element name="p">
  <attribute name="a2"></attribute>
  <element name="c1"><text/></element>
  <attribute name="a1">
  <choice><value>a</value><value>b</value></choice>
  </attribute>
  <element name="c2"><empty/></element>
</element>
```

Choice Between Elements and Attributes

The uniformity of syntax yield benefits in expressive power: we can naturally express a choice between an element and an attribute:

```
<define name="Email"> <!-- can be attribute of parent, element, or structure -->
  <choice>
    <attribute name="email"><text/></attribute>
    <element name="email">
       <choice>
          <text/>
          <group>
            <element name="userid"><text/></element>
            <element name="host"><text/></element>
          </group>
       </choice>
    </element>
  </choice>
</define>
```

The one difference between element and attribute definitions is that an attribute definition can be an empty element, and its content will default to `<text/>`;. The following two definitions are equivalent:

```
<attribute name="email"><text/></attribute>
<attribute name="email"></attribute>
```

An element definition cannot be empty: it has to explicitly say `<text/>` or `<empty/>`.

Lists

In DTDs, only attribute values (not element content) can be defined as a list of tokens. RELAX NG again provides uniform treatment. We show here an element definition, but an attribute definition would use exactly the same syntax:

```
<element name="listTest">
  <list>
    <oneOrMore>
      <data type="token"/>
    </oneOrMore>
  </list>
</element>
```

This will match such content as

```
<listTest>a b c d</listTest>
```

If we made a data type library available (see the section “The datatypeLibrary and XML Schema Data Types” later in this chapter), we could declare an element type that is a list of integers or Booleans. Even a list of tokens is different enough from string to make it worth defining: each item on the list must conform to the XML name production, and whitespace is normalized in a list of tokens but not in a string.

Interleaving of Elements and the Mixed-Content Model

In DTDs, there is no way to define an unordered set of elements, short of writing out the disjunction of all possible reorderings. RELAX NG provides the `interleave` pattern for this purpose. You can interleave any patterns, including optional elements, `zeroOrMore` patterns, and `oneOrMore` patterns.

The `mixed` pattern, corresponding to the DTD mixed-content model, is, in fact, shorthand for interleaving text with elements:

```
<mixed>somePattern</mixed>
```

stands for

```
<interleave> <text/> somePattern </interleave>
```

Unlike DTDs, this allows the order of children to be specified even within the mixed-content model. Consider a document that introduces many new terms

that each must be followed by a definition, perhaps separated by some text. The following grammar (termDef.rng) enforces that constraint. (See Listing 8-6.) The grammar filters out documents in which a `definition` element appears before the matching `term` element, or one of the two is missing.

Listing 8-6. Terms and Definitions

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
<start>
  <element name="termdef">
    <ref name="termdefContent"/>
  </element>
</start>
<define name="termdefContent">
  <mixed>
    <oneOrMore>
      <ref name="TermRef"/><ref name="DefRef"/>
    </oneOrMore>
  </mixed>
</define>
<define name="TermRef">
  <element name="term"><data type="token"/></element>
</define>
<define name="DefRef">
  <element name="def"><data type="string"/></element>
</define>
</grammar>
```

This grammar will accept termDef.xml in Listing 8-7, but reject the same file with a `term` and a `def` switched around or one of them missing.

Listing 8-7. Document with Terms and Definitions

```
<termdef>
Some text 1
<term>a</term>
More text 1
<def>first letter</def>
Some text 2
<term>b</term>
More text 2
<def>second letter</def>
Even more text
</termdef>
```

Namespace Support

RELAX NG supports namespaces. A name defined by a RELAX NG grammar consists of a namespace URI and a local NCName (no-colon name). It matches a name in the instance document only if both the namespace URI and the local name match. A RELAX NG processor matches names against definitions in a context that contains all the defined mappings from prefixes to namespace URIs.

To specify the namespace of a name being defined, the `ns` attribute is used:

```
<element name="e" ns="http://www.n-topus.com/ns/rngexample">
  <text/>
</element>
```

This will match any of the three following elements, but no names from a different namespace:

```
<e xmlns="http://www.n-topus.com/ns/rngexample">hmmmm. . . . </e>
<p:e xmlns:p="http://www.n-topus.com/ns/rngexample">well, maybe</p:e>
<pre:e xmlns:pre="http://www.n-topus.com/ns/rngexample">any text, really</pre:e>
```

The `ns` attribute can appear on any element of a RELAX NG grammar, and the namespace it specifies will attach itself to every name defined within its scope. So, if you specify a namespace on a `choice` element:

```
<choice ns="http://www.n-topus.com/ns/rngexample">
  <element name="na">...</element>
  <element name="nb">...</element>
  <element name="nc">...</element>
</choice>
```

all the element names defined within the `choice` element will belong to that namespace.

If all the names in instance documents come from the same namespace, it makes sense to put the `ns` attribute on the `grammar` element. The effect is that every time you say

```
<element name="..."
```

the name you have defined will belong to the namespace specified on the `grammar` element.

QNames in Definitions

What if your instance documents combine vocabularies from more than one namespace? One solution is to repeat `ns` attributes within the grammar every time you switch from one vocabulary to another. This is tedious and prone to errors, so RELAX NG allows you to associate the "target" namespaces with prefixes (using the standard `xmlns` attribute) and to use qualified names as the values of the name attribute. For instance, suppose that your instance documents combine Resource Definition Framework and Dublin Core vocabularies. You can start your grammar like this:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
...
>
```

The following two definitions will define names in two different namespaces:

```
<element name="rdf:Description">...</element>
<element name="dc:description>...</element>
```

An alternative approach is to define separate grammars for the two vocabularies and use the `include` element to incorporate them into your combined vocabulary. We will show how to do that in the section on modularity and reuse.

Namespaces and Attributes

Attributes, as you know, do not inherit the namespace specified on its parent element but rather default to no namespace (that is, the Namespace URI that is the empty string). Suppose that you say

```
<element name="e" ns="http://www.n-topus.com/ns/rngexample">
  <empty/>
  <attribute name="a"><text/></attribute>
</element>
```

This is equivalent to

```
<element name="e" ns="http://www.n-topus.com/ns/rngexample">
  <empty/>
  <attribute name="a" ns=""><text/></attribute>
</element>
```

Both will match the first two of the following elements but not the third one:

```
<e xmlns="http://www.n-topus.com/ns/rngexample" a="whatever"/>
<p:e xmlns:p="http://www.n-topus.com/ns/rngexample" a="whatever"/>
<p:e xmlns:p="http://www.n-topus.com/ns/rngexample" p:a="whatever"/>
```

To have attribute names in a namespace, you have to use one of the following approaches:

- Specify the namespace in the attribute definition using the `ns` attribute.
- Associate the namespace with a prefix and use a QName in the attribute definition.
- Define the "namespaced" attributes in a separate grammar and include it in the grammar for the combined vocabulary.

We will illustrate the last approach in the "rddl.rng" section later in the chapter where we define a language that combines the XHTML and XLink vocabularies.

Namespaces Within the Grammar: Annotations

RELAX NG grammars are themselves XML documents that use names from a specific namespace. All names from other namespaces are ignored by the validator and therefore can serve as annotations or comments. For instance:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:ann="http://www.n-topus.com/ns/rngexample/"
...
>
<ann:comment>This grammar is for a bibliographic database language
 that combines two RDF elements with the Dublin Core metadata vocabulary.
Arbitrary XHTML (except applets and objects) can be used within
dc:description elements</ann:comment>
```

Suppose that you have another XML vocabulary within a namespace that is used to write programs for some processor to execute. You can intersperse a text in that vocabulary with the RELAX NG grammar, extract it using XSLT, and run the other program. This technique has been suggested for Schematron patterns

embedded in XML schemas (see
`http://www.ascc.net/xml/resource/schematron/schematron.html`), but it can also be used with RELAX NG grammars.

An Annotation Vocabulary for DTD Compatibility

To facilitate transition from DTDs to RELAX NG, a separate specification defines a controlled annotation vocabulary to express two features of DTDs that are not expressible in RELAX NG grammars. These features are default attributes and the validity constraints on the ID and IDREF attribute data types. (IDs must be unique within a document; IDREFs must refer to an existing ID.) Consider this DTD (from the specification):

```
<!DOCTYPE employees [
<!-- A list of employees. -->
<!ELEMENT employees (employee*)>
<!-- An individual employee. -->
<!ELEMENT employee (#PCDATA)>
<!ATTLIST employee
  id ID #REQUIRED
  manages IDREFS #IMPLIED
  managedBy IDREF #IMPLIED
  country (US|JP) "US"
>
]>
```

It can be translated to the following RELAX NG schema (with annotation-related material highlighted):

```
<element name="employees"
    xmlns="http://relaxng.org/ns/structure/1.0"
    xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
    datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0">
  <a:documentation>A list of employees.</a:documentation>
  <zeroOrMore>
    <element name="employee">
      <a:documentation>An individual employee.</a:documentation>
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <optional>
        <attribute name="manages">
          <data type="IDREFS"/>
```

```
        </attribute>
      </optional>
      <optional>
        <attribute name="managedBy">
          <data type="IDREF"/>
        </attribute>
      </optional>
      <optional>
        <attribute name="country" a:defaultValue="US">
          <choice>
            <value>US</value>
            <value>JP</value>
          </choice>
        </attribute>
      </optional>
      <text/>
    </element>
  </zeroOrMore>
</element>
```

As you can see, the DTD compatibility package provides the following features:

- `documentation` element within the “compatibility” namespace
- `defaultValue` attribute within the same namespace
- a data type library that defines the ID, IDREF, and IDREFS types as they are defined in *XML 1.0*

The specification also defines conformance requirements for processors that implement the compatibility package. As of this writing (February 2002), neither Jing nor MSV implements it.

Name Classes

Normally, the `element` pattern (that is, the `element` element in the RELAX NG namespace) has a `name` attribute, and matches only those elements in instance documents that have the same name as the value of that attribute. However, this is too rigid for some tasks. Consider RDDL that we covered in Chapter 2: it is XHTML Basic augmented by one element called `resource`. As currently defined, resource can use only simple links and, therefore, five specific XLink attributes.

(We will define RDDL according to its current spec in the "rddl.rng" section later in this chapter.) Suppose we wanted to say that `resource` (or some other element in some other language) can take all XLink attributes. We would like to be able to say that `resource` can take any attribute from the XLink namespace. RELAX NG allows you to do that, using the `nsName` element, as in the highlighted line in the following code:

```
<element name="resource"><!-- first we define resource element -->
  <ref name="resource.attlist"/>
  <ref name="Flow.model"/><!-- defined in XHTML Basic -->
</element>
<define name="resource.attlist">
  <ref name="lang.attrib"/><!-- defined in XHTML Basic -->
  <ref name="id.attrib"/><!-- defined in XHTML Basic -->
  <ref name="XMLBASE.attrib"/><!-- defined in XHTML Basic -->
  <nsName ns="http://www.w3.org/1999/xlink"/><!-- any XLink attribute -->
</define>
```

You will see these definitions in a larger context in the section on modularity, in which we'll show you the entire modularization framework implemented in RELAX NG. For now, just note the single highlighted line in which a name class is used.

The `nsName` element can be used without an `ns` attribute, in which case its value defaults as usual to the element's inherited namespace.

Combining Name Classes

You can perform set operations on name classes. The `choice` name class matches any name that is matched by any of its children name classes. The `not` name class matches all names that are not matched by its child name class. The following example (adapted from the tutorial) shows an element that can have any number of attributes from any namespace other than its own:

```
<element name="cite" ns="http://www.example.com">
  <zeroOrMore>
    <attribute>
      <not>
        <choice>
          <nsName ns=""/> <!-- must be in a namespace -->
          <nsName/> <!-- the namespace must be different from its element's -->
        </choice>
      </not>
```

```
    </attribute>
  </zeroOrMore>
  <text/>
</element>
```

By itself, the highlighted part matches a single attribute (from an open set of names); to match any number of those, the `zeroOrMore` pattern is needed.

Modularity and Reuse

The main tools of modularization are two patterns: `externalRef` and `include`. Any pattern residing in an external file can be referenced from a RELAX NG grammar using the `externalRef` pattern. A grammar residing in an external file can be included in another grammar using the `include` pattern. Both elements have a required `href` attribute and are processed in the same way:

1. The XML text referenced by the `href` attribute is parsed.

2. All attributes from the referring element (`externalRef` or `include`) are transferred to the result of parsing.

3. The result of parsing replaces the referring element.

This provides a mechanism for creating vocabularies out of reusable modules.

Combining and Replacing Definitions

In addition to reusing modules, we want to be able to reuse (that is, redefine) names. RELAX NG allows multiple definitions of the same name and provides three ways of combining them: choice, interleave, and replace.

> **NOTE** *Multiple definitions usually result from an external reference or inclusion, but there is no prohibition against a single grammar having multiple definitions of the same name. The mechanisms for combining multiple definitions are the same in both cases.*

Combining Definitions

If a grammar contains multiple definitions of the same name, at least one of them must have a `combine` attribute. The possible values of `combine` are `choice` or `interleave`. The effect is equivalent to wrapping alternative definitions into a `choice` or `interleave` pattern. In both cases, the order in which alternatives appear does not matter: the `choice` and `interleave` operations are order independent (or *commutative*, in algebraic terms). In the DTD framework, as we saw in last chapter, combining is done by concatenating strings, which is not commutative. In general, commutative operations are better because they remove a big burden from the programmer and a frequent source of bugs.

Replacing Definitions

In the case of included grammars, definitions in the including grammar can also be replaced by new definitions. The new definitions must be supplied as children of the `include` element. Three element types can appear, in any order and any number of times, as children of `include`: `define`, `start`, and `div`. A `define` child of `include` replaces the definition(s) of the same name within the included grammar, and a `start` child of `include` redefines its `start` element, if any. The `div` element has the same content model as `grammar` and serves to group definitions together.

As an example of modularity and reuse, consider how XHTML modularization and XHTML Basic are implemented in RELAX NG. You may want to review the XHTML Basic section of Chapter 3, to be reminded of the issues.

XHTML Modularization

The current version of RELAX NG implementation of XHTML modularization can be found at `http://thaiopensource.com/relaxng/xhtml/`. The home page says, in part,

> *The modules directory contains all the modules; xhtml-basic.rng uses the modules to implement XHTML Basic; xhtml-strict.rng uses the modules to implement XHTML 1.0 strict; xhtml.rng uses the modules to implement the union of XHTML 1.0 transitional and XHTML 1.0 frameset. . . . To create a custom version of XHTML, simply copy xhtml.rng and delete the inclusions of the modules that you do not want.*

The biggest difference between the RELAX NG implementation and the DTD implementation is that the RELAX NG implementation does not require you to create a model module specific to the combination of XHTML modules you are using. Instead, simply include the modules you want. The modules take care of redefining the content models appropriately.

Let's start by looking at xhtml.rng, which corresponds to the DTD driver file.

xhtml.rng

This file contains nothing but `include` patterns, providing, in effect, a listing of all available modules. (See Listing 8-8.)

Listing 8-8. The XHTML Driver File for the Union of Transitional and Frames DTDs

```
<!-- This corresponds to the union of transitional or frameset DTDs -->
<grammar ns="http://www.w3.org/1999/xhtml"
         xmlns="http://relaxng.org/ns/structure/1.0">
<include href="modules/datatypes.rng"/>
<include href="modules/attribs.rng"/>
<!-- <include href="modules/struct.rng"/> -->
<include href="modules/frames.rng"/>
<include href="modules/text.rng"/>
<include href="modules/hypertext.rng"/>
<include href="modules/list.rng"/>
<include href="modules/image.rng"/>
<include href="modules/ssismap.rng"/>
<include href="modules/base.rng"/>
<include href="modules/link.rng"/>
<include href="modules/meta.rng"/>
<include href="modules/param.rng"/>
<include href="modules/object.rng"/>
<include href="modules/bdo.rng"/>
<include href="modules/pres.rng"/>
<include href="modules/edit.rng"/>
<include href="modules/applet.rng"/>
<!-- <include href="modules/basic-form.rng"/> -->
<include href="modules/form.rng"/>
<include href="modules/style.rng"/>
<include href="modules/script.rng"/>
<!-- <include href="modules/basic-table.rng"/> -->
<include href="modules/table.rng"/>
```

```
<include href="modules/csismap.rng"/>
<include href="modules/events.rng"/>
<include href="modules/inlstyle.rng"/>
<include href="modules/target.rng"/>
<include href="modules/iframe.rng"/>
<include href="modules/nameident.rng"/>
<include href="modules/legacy.rng"/>
</grammar>
```

You will notice that three modules are commented out: Basic-form, Basic-table, and Struct. The two Basic modules are commented out because they are included in their full-sized siblings. Unlike the DTD version, in which the Forms and Table modules are written from scratch because, without any code reuse, the RELAX NG implementation derives them from the Basic modules. We will see how it is done after we review frames.

The Frames Problem

To remind you of the situation with frames, we have to go back all the way to the *HTML 4.0* recommendation. It contains three SGML DTDs: strict, transitional, and loose, in addition to a special frameset DTD, in which the `html` element consists of `head` and `frameset`, rather than `head` and `body`. The loose DTD was intended to justify and forgive many "features" that had been introduced before HTML was codified. The transitional DTD was less forgiving, and it intended to provide a migration path to the strict DTD. The frameset DTD was supposed to complement the loose and transitional ones but not the strict DTD. The strict DTD disallowed many features that were listed as "deprecated" in transitional DTD; it also severed any connection with the frameset and frame elements. (In the world of strict DTD, you are supposed to achieve this kind of effect with stylesheets and positioning.) *XHTML 1.0* dropped the loose DTD but carried over the strict and transitional ones. It also preserved the frameset DTD, in combination with the transitional DTD but not with the strict one.

As James Clark (and probably others) have observed, there is really no reason to separate frameset and frame into a special DTD of their own because one can easily say

```
<!ELEMENT html (head, (body|frameset))>
```

The reasons why this has not been done are historical and probably irrational, but even the modularized XHTML continues to carry a separate Frames module. RELAX NG gets rid of it: there is a Struct module that defines `html` as `(head,body)` and a Frames module that defines `html` as `(head, (body|frameset))`.

The driver file of Listing 8-8 uses the Frames module and comments out the Struct module; the driver files for the strict XHTML and for XHTML-Basic comment out the Frames module and use the Struct module instead. Because the Stuct module is more constrained, it is reused in the Frames module, as explained in the next section.

The Reuse of the Struct Module

The Struct module defines the `html`, `head`, and `body` elements. It is used in the Strict and Basic grammars because they don't allow frames. In the transitional version (in which frames are allowed), the Struct module is reused within the Frame module, which redefines `html` to allow a choice between `body` and `frameset`. The definition of `body` is taken unchanged from the Struct module; the Frames module adds the definition of `frameset`. Here is the beginning of the Frames module, in which `html` is redefined (in the highlighted lines of code):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
<include href="struct.rng">
  <define name="html">
    <element name="html">
      <ref name="html.attlist"/>
      <ref name="head"/>
      <choice>
            <ref name="body"/>
            <ref name="frameset"/>
      </choice>
    </element>
  </define>
</include>
```

As mentioned in the preceding section, `include` elements can have `define` children that replace definitions of the same names in the included grammar. This is how `html` is redefined here. The `body` element is available from the Struct module, and `frameset` is defined in the rest of this grammar. In the end, the `noframes` element again references `body`:

```
<define name="noframes">
  <element name="noframes">
    <ref name="noframes.attlist"/>
    <ref name="body"/>
  </element>
</define>
```

```
<define name="noframes.attlist">
  <ref name="Common.attrib"/>
</define>
</grammar>
```

`Common.attrib` is defined in the Attribs module, attrib.rng. The Attribs and Datatype modules are included in every XHTML-based language.

Basic and Non-Basic Modules for Forms and Tables

Further examples of code reuse are found in the Form and Table modules, which include their Basic cousins. We will compare form.rng and basic-form.rng. In the Basic module, the form element is defined as

```
<define name="form">
  <element name="form">
    <ref name="form.attlist"/>
    <ref name="Block.model"/>
  </element>
</define>
```

wherein `Block.model` is defined, in the Text module, as

```
<define name="Block.model">
  <oneOrMore>
    <ref name="Block.mix"/>
  </oneOrMore>
</define>
```

and `Block.mix` is defined, also in the Text module, as

```
<define name="Block.mix">
  <ref name="Block.class"/>
</define>
```

> **NOTE** *The reason that* `Block.class` *is wrapped into* `Block.mix` *is because the Legacy module redefines* `Block.mix` *to include inline elements. It would be contrary to the framework's naming conventions to call the result* `Block.class`.

Finally, the Text module defines `Block.class` as a choice of text containers:

```
<define name="Block.class">
  <choice>
    <ref name="address"/>
    <ref name="blockquote"/>
    <ref name="div"/>
    <ref name="p"/>
    <ref name="pre"/>
    <ref name="Heading.class"/><!-- h1 through h6 -->
  </choice>
</define>
```

We have reached the bottom but have not found the control elements of the form, such as `input` or `textarea`. As you can guess, `Block.class` gets redefined in the Form module:

```
<define name="Block.class" combine="choice">
  <ref name="Form.class"/>
</define>
```

This makes it possible for forms to contain forms, together with their control elements.

The non-Basic Form module includes the Basic Form module and redefines the `form` and `select` elements, to allow `fieldsets` in forms and `optgroups` in `select` elements:

```
<include href="basic-form.rng">
  <define name="form">
    <element name="form">
      <ref name="form.attlist"/>
      <oneOrMore>
            <choice>
              <ref name="Block.mix"/>
              <ref name="fieldset"/>
            </choice>
      </oneOrMore>
    </element>
  </define>

  <define name="select">
    <element name="select">
      <ref name="select.attlist"/>
```

```
        <oneOrMore>
              <choice>
                <ref name="option"/>
                <ref name="optgroup"/>
              </choice>
        </oneOrMore>
      </element>
    </define>
  </include>
```

We are ready to try an example of our own: a RELAX NG grammar for RDDL. It is similar but not identical to Jonathan Borden's grammar at `http://www.openhealth.org/RDDL/xhtml-rddl.rng`.

rddl.rng

RDDL, as you recall, is just XHTML Basic with an additional `resource` element. The content model of `resource` is arbitrary XHTML Basic. The attributes of `resource` consist of three common attributes (`id`, `xml:base`, and `xml:lang`) and five XLink attributes (`type`, `href`, `title`, `role`, and `arcrole`). In the current version of RDDL, only simple links are allowed. The rddl.rng grammar is a driver file that includes all XHTML Basic modules plus two additional modules that we have to define: a Resource module and an XLink module.

Resource Module

The Resource module is in resource.rng (Listing 8-9).

Listing 8-9. resource.rng

```
<grammar
  ns="http://www.rddl.org/"
  xmlns="relaxng.org/ns/structure/1.0">
<define name="resource">
  <element name="resource">
    <ref name="resource.attlist"/>
    <ref name="Flow.model"/>
  </element>
</define>
<define name="resource.attlist">
  <ref name="lang.attrib"/>
  <ref name="id.attrib"/>
```

```
  <ref name="XLINK.simple.attlist"/>
</define>
</grammar>
```

We declare the target namespace to be `http://www.rddl.org`, and, within that namespace, we declare the resource element. Its content mode is `Flow.model`, as defined in text.rng: a choice of all block and inline elements of XHTML. Its attributes are

- `lang.attrib` and `id.attrib`, defined in the Attribs module of XHTML Basic
- `XLINK.simple.attlist`, defined in the additional XLink module

XLink Module

The XLink module is in xlink.rng. It defines attributes and groups of attributes, as shown in Listing 8-10.

Listing 8-10. xlink.rng

```
<!-- XLink Module -->
<grammar ns="http://www.w3.org/1999/xlink"
                  xmlns="http://relaxng.org/ns/structure/1.0"
                  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
>
<define name="XLINK.type.attrib">
  <attribute name="type" ns="http://www.w3.org/1999/xlink">
    <choice>
      <value>simple</value>
      <value>extended</value>
      <value>arc</value>
      <value>locator</value>
      <value>resource</value>
    </choice>
  </attribute>
</define>

<define name="XLINK.href.attrib">
 <optional>
  <attribute name="href" ns="http://www.w3.org/1999/xlink">
    <data type="anyURI" />
  </attribute>
  </optional>
</define>
```

```
<!-- XLINK.role.attrib, XLINK.arcrole.attrib and XLINK.title.attrib
are defined in the same way -->

<define name="XLINK.simple.attlist" combine="choice">
  <ref name="XLINK.type.attrib"/>
  <ref name="XLINK.href.attrib"/>
  <ref name="XLINK.role.attrib"/>
  <ref name="XLINK.arcrole.attrib"/>
  <ref name="XLINK.title.attrib"/>
</define>
</grammar>
```

These two modules—resource.rng and xlink.rng—are all that is needed to define RDDL as an extension of XHTML Basic. It is instructive to compare them with the RDDL DTD at `http://www.openhealth.org/RDDL/rddl-xhtml.dtd`; the difference in complexity is quite striking.

The new RELAX NG material in xlink.rddl consists of the `datatapeLibrary` attribute and the `anyURI` data type. These are explained in the next section.

The `datatypeLibrary` and XML Schema Data Types

RELAX NG defines two native data types: string and token. If any other data type is used in a grammar, the processor will try to look it up in an external data type library. If no such library is specified, the unknown type is equivalent to `string`. (Some users might prefer to see an error message in this situation.)

An external library is specified by the `datatypeLibrary` attribute, whose value is a URI that identifies the library. The attribute can be specified on any element, and its value is inherited by the element's descendants in the same way that namespace declarations are inherited. An obvious place for the `datatypeLibrary` attribute is the grammar element, as in Listing 8-10.

XML Schema Part 2, Built-in Types

The only external library currently available, and one that is likely to be standard, is *XML Schema Part 2*. The library contains a great number of built-in simple types—some of them primitive, others derived. For instance, the primitive type `decimal` represents arbitrary precision decimal numbers; the built-in type `integer` is derived from `decimal` by setting the number of fraction digits to 0. From integer, a number of other integer types are derived, such as `long`, `int`, `short`, and `byte`. All these derivations are by restriction, but you can also derive new types by forming a list of simple types. This is how the `IDREFS` type is derived

from IDREF. A complete diagram of built-in data types (with primitive and derived types color-coded and different kinds of derivation shown with different kinds of lines) can be found at `http://www.w3.org/TR/xmlschema-2/#built-in-datatypes`.

The Jing processor comes with support (currently not quite complete) for that library. Specifically, the current version supports the built-in XML Schema data types in the following list, which is quoted from the Jing data types Web page `http://thaiopensource.com/relaxng/jing-datatypes.html` but rearranged and divided into sublists:

- *string types*: string normalizedString token
- *numeric types*: decimal float double
- *integer types*: integer nonPositiveInteger negativeInteger long int short byte nonNegativeInteger unsignedLong unsignedInt unsignedShort unsignedByte positiveInteger
- *XML name types*: Name QName NCName
- *XML attribute types*: NOTATION ID IDREF IDREFS NMTOKEN NMTOKENS
- *self-explanatory types:* language, Boolean

The following clarifications and caveats are also from the Jing data types page:

- Other data types (such as `anyURI`) are recognized but validated as strings.
- `ID` and `IDREF` are validated as tokens, and `IDREFS` as a list of tokens. Jing does not check the validity constraints on these attributes, but it will once it implements the DTD compatibility library. (See the section on annotations earlier in the chapter.)
- The `NOTATION` attribute is validated simply as a QName. In *XML 1.0*, there are additional validity constraints on its use, but it is practically never used, either in DTDs or XML schemas.

In addition to built-in types, XML Schema offers a mechanism for creating user-defined derived types. The mechanism, also implemented by Jing, is presented in the next section, which is a brief excursion into the world of XML Schema.

XML Schema Part 2, User-Defined Types

The notion of a data type is quite precisely defined in XML Schema: it is a triple consisting of a set of values, a set of literals to represent those values, and a set of facets. The set of values is called the *value space*, and the set of literals the *lexical space*. Within the lexical space, there may be multiple representations for the same values, in which case a subset of the lexical space is designated as the *canonical representation*. For instance, the value space of Boolean consists of two values {true, false}, its lexical space of legal literals is {true, false, 1, 0}, and its canonical representation is the set of literals {true, false}.

double

To take a more elaborate example that will also give us interesting facets to talk about, the value space of the `double` data type is the set of values of the form $m \times 2^e$, in which m is an integer whose absolute value is less than 2^{53}, and e is an integer between –1075 and 970, inclusive. In addition, the value space contains positive and negative 0 (literals 0 and –0), positive and negative infinity (literals `inf` and `-inf`) and a special not-a-number value (literal `NaN`). This value space is known as "the IEEE double-precision 64-bit floating-point type [IEEE 754-1985]", which is familiar from Java and JavaScript; it is also the one used by XSLT.

The lexical representations of doubles are decimal numbers or decimal numbers in the scientific notation, with a mantissa followed (optionally) by "e" or "E", followed by an exponent that must be an integer. If there is no exponent, 0 is assumed. The canonical representation has a mantissa with a single digit before the decimal point and at least one digit after the decimal point; the exponent is required.

Facets

Facets are properties. There are *fundamental facets* and *constraining facets*. Fundamental facets are the same for all data types. The five facets are: equal, ordered, bounded, cardinality, and numeric. They describe the value space: Is equality defined on it? (The answer is yes, for all of them.) Is it ordered? Is it bounded? Is it finite or infinite? Is it numeric? For instance, Boolean is not ordered, not bounded (because it's not ordered), not numeric, and finite; double is totally ordered, bounded, numeric, and finite. There isn't much to say about fundamental facets; they are just there.

> **NOTE** *A complete table of XML Schema built-in data types and the values of their fundamental facets can be found at* `http://www.w3.org/TR/xmlschema-2/#section-Datatypes-and-Facets.`

Constraining facets are for deriving data types. For instance, the constraining facets of double are

- pattern
- enumeration
- whitespace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

The whitespace facet is something of a black sheep in this list: instead of constraining the set of values, it controls the whitespace handling. Its possible values are preserve, replace, and collapse: *preserve* means “leave as is,” *replace* means “replace every whitespace character with #x20,” and *collapse* means “collapse runs of whitespace characters into a single #x20 character.” The rest of the facets are discussed and illustrated in the following sections.

Examples of XML Schema Type Definitions

Listing 8-11 provides some examples of simple type definitions in the XML Schema language.

Listing 8-11. Examples of XML Schema Simple Type Definitions

```
<simpleType name="doubleRangeType">
  <restriction base="double">
   <maxInclusive value="5" />
   <minInclusive value="2" />
  </restriction>
</simpleType>
```

```
<simpleType name="doubleEnumType">
  <restriction base="double">
     <enumeration value="3.1"/>
     <enumeration value="4.7"/>
     <enumeration value="5.3"/>
  </restriction>
</simpleType>

<simpleType name="doubleListType">
  <list itemType="double"/>
</simpleType>
```

With types so defined, you can define elements as follows:

```
<element name="doubleInRange" type="doubleRangeType"/>
<!-- 2.0, 3.7, 4.99, 5.0 are legal values; 1.5, 5.001 are not -->
<element name="doubleEnumValue" type="doubleEnumType"/>
<!-- 3.1, 4.7, 5.3 are the only legal values -->
<element name="doubleList" type="doubleListType"/>
<!-- legal values are whitespace separated lists of doubles -->
```

With elements and types so defined, elements in instance documents will be checked for type conformance by a Schema validator. The motivation is that application programs will not have to do type checking themselves.

Anonymous Types

Instead of defining a named type and using its name in defining an element, you can have an anonymous type defined within the element definition:

```
<element name="doubleList">
  <simpleType>
    <list itemType="double"/>
  </simpleType>
</element>
```

The difference is that, if you give a type a name, you can reuse it, by reference, in other element definitions.

Non-Atomic Simple Types

List types are simple types that are not atomic. Another kind of non-atomic simple types are union types. Suppose that you want to be able to specify a font size either as an integer in the range from 8 to 72 or as one of three string tokens: `small`, `medium`, and `large`. You can define the corresponding type and element as shown in Listing 8-12.

Listing 8-12. Union Type

```
<simpleType name="FontSizeType">
  <union>
   <simpleType>
     <restriction base="positiveInteger">
        <minInclusive value="8"/>
        <maxInclusive value="72"/>
     </restriction>
   </simpleType>
   <simpleType>
     <restriction base="NMTOKEN">
       <enumeration value="small"/>
       <enumeration value="medium"/>
       <enumeration value="large"/>
     </restriction>
   </simpleType>
  </union>
</simpleType>
<element name="FontSize" type="FontSizeType" />
```

In Summary

In summary, simple data types of XML Schema fall into these three categories:

- primitive or derived
- built-in (primitive or derived) or user-defined (derived). The only way to create a new primitive type is by revising the W3C recommendation.
- atomic or non-atomic (list, union). There are built-in list types (NMTOKENS), but not union types.

Learning the simple type system is not hard because most types are familiar either from programming languages or from *XML 1.0*. If you want to create your

own types, then, in addition to built-in types, you need to know the facets. Here is the complete list of constraining facets, divided into groups of similar meaning:

- length, minLength, maxLength
- minInclusive, minExclusive, maxInclusive, maxExclusive
- pattern (Regular Expressions)
- enumeration
- duration, period (for time-based types)
- encoding (hex or base64)
- scale (number of digits in fractional part)
- precision (number of significant digits)
- whitespace (one of: preserve, replace, collapse)

The Pattern Facet and Regular Expressions

The most important facet, by far, is the pattern facet that allows you to specify a regular expression to constrain the values of the type. Regular Expressions, in case you have not used them before, form a language for specifying sets of characters and strings of characters. They are used in the context of pattern matching: a regular expression forms a pattern against which strings are matched. Here are a few examples:

- Most individual characters match themselves: the chararacter "-" matches itself.
- The pattern "\d" matches any digit. The pattern "\d{3}" matches any sequence of three digits.
- The pattern 315-\d{3}-\d{4} matches any telephone number in the 315 area code of the United States, in the 315-123-4567 format.

- The pattern \(\d{3}\)\d{3}-\d{4} matches any telephone number in the United States, in the (315)228-7719 format. (You have to escape the "(" and ")" characters because they have a special meaning in the Regular Expression language.)

- The pattern (\(\d{3}\)|\d{3}-)\d{3}-\d{4} matches any telephone number in the United States in either of the two formats. (The unescaped "(" and ")" are used for grouping; the "|" character means "or".)

Regular Expressions are a big topic, and whole books are written on them. The entire Perl programming language is built around Regular Expressions. Regular Expressions are also an important part of JavaScript, in addition to being an important part of *XML Schema Part 2*, which has a large appendix on Regular Expressions. The new feature of Regular Expressions as used in XML Schema is that they include expressions for classes of Unicode characters. (Until recently, Regular Expressions covered only ASCII.) It is a measure of XML Schema's size and ambition that it includes, as a brief aside, a fifteen-page specification for Unicode Regular Expressions.

Using Regular Expressions to Define Simple Types

To define a simple type for U.S. telephone numbers in the 315-123-4567 format, use the pattern facet:

```
<simpleType name="USPhoneType">
  <restriction base="string">
   <pattern value="\(\d{3}\)\d{3}-\d{4)"/>
  </restriction>
</simpleType>
```

The pattern facet can be used with almost any base type, including numeric types. You could define a range of integers from 23 to 76 by saying

```
<simpleType name="intRange">
  <restriction base="integer">
   <pattern value="2[3-9]|[3-6][0-9]|7[0-6]"/>
  </restriction>
</simpleType>
```

The pattern reads

- a 2 followed by digit in the range 3-9 *or*
- a digit in the 3–6 range followed by a digit in the 0–9 range (that is, any digit, we could have used "\d" as well) *or*
- a 7 followed by a digit in the range 0–6 range

There is a tool that automatically generates such patterns from integer and decimal ranges: `http://www.xfront.com/#regexGen`. The latest versions of Xerces (1.3.1 or later) include a library for processing Regular Expressions, as part of its XML-Schema evaluator. Jing also uses this library.

Validating Derived Types with Jing

To validate derived simple types, RELAX NG allows data elements to have `param` children that have a `name` attribute whose value is a facet name. The content of the `param` element is the value of the facet. For instance, to constrain a data type to strings of length 31 or less, we say

```
<element name="SMS">
  <data type="string">
    <param name="maxLength">31</param>
  </data>
</element>
```

As of this writing (but check the current Jing documentation!), to validate Regular Expression patterns (the `pattern` facet), you have to use the Java version and specifically the Xerces XML parser because `xerces.jar` includes the Regular Expressions processor. (For Java programmers, the processor is in the `org.apache.xerces.utils.regex` package.) To run Jing with that processor enabled, you set the value of a System property, `com.thaiopensource.relaxng.util.RegexEngine`, to the corresponding class. If you run Jing from the command line (or, more likely, from the jingx batch file), you would insert this option (divided into two lines):

```
-Dcom.thaiopensource.relaxng.util.RegexEngine=
com.thaiopensource.relaxng.util.XercesRegexEngine
```

This concludes our discussion of *XML Schema Part 2* data types and how they are used as a data-typing library with RELAX NG. We are now going to

attempt an overview of *XML Schema Part 1*, concentrating on those aspects of it that we'll put to use in the next chapter.

XML Schema Part 1: Structures

We will start with a simple schematic example and a comparison with RELAX NG. In the context of the example, we will also develop a framework for testing. After this initial round of piecemeal discovery, we will provide a more systematic review.

To avoid clutter, in this section, we will abbreviate *XML Schema Part 1* as *XS1*, and *XML Schema Part 2* as *XS2*. We will also refer to the corresponding W3C recommendations as *XS1R* and *XS2R* (frequently followed by a section number).

We will make occasional references—in addition to the recommendations—to a very useful set of materials developed by Roger Costello (`xfront.com`). In particular, we will use his XML Schema Tutorial at `http://www.xfront.com/#schema`, and XML Schema Best Practices (abbreviated to *xsBest*) at `http://www.xfront.com/BestPracticesHomepage.html`.

An Example and a Comparison

Consider a simple document: *r* stands for *root*, *c* stands for *child*, and *gc* stands for *grandchild*. Every grandchild is of a specific type: string, a string pattern, a number, and a year. They can be thought of a part name, part stock number, current price, and the year it was added to the product line, as shown in Listing 8-13.

Listing 8-13. Simple Schematic Document, xs/xsEx1.xml

```
<?xml version="1.0"?>
<r>
  <c>
   <gcStr>any string</gcStr>
   <gcStrPat>22-abc-z12</gcStrPat><!-- a string pattern -->
   <gcNum>123.45</gcNum><!-- a number -->
   <gcYear>1999</gcYear><!-- a year -->
  </c>
  <!-- many more children -->
</r>
```

Proceeding from the bottom up, consider the `gcNum` element. In RELAX NG, it can be declared without preserving data type information:

```
<element name="gcNum"><text/></element>
```

Alternatively, a data-typing library (which may or may not be XS2) can be invoked and the data type information included in the declaration:

```
<!-- in scope for datatypeLibrary declaration -->
 <element name="gcNum">
  <data type="decimal"/>
</element>
```

In XML Schema, you don't have these choices, you have to say

```
<element name="gcNum" type="..." />
```

You can, of course, specify the type as `string` or as `decimal`, but there has to be a type specification, and that specification must be a built-in type defined in XS2, or a type derived from a built-in type by the rules specified in XS2. Moreover, XS1 requires that these data type specifications must be present in the output of a conformant XS processor, the so-called *Post-Schema-Validation Infoset*, or *PSVI*.

Validation, Assessment, and PSVI

XS1R, Section 2.4 ("Conformance") defines three levels of conformance, beginning with the minimal one:

> *[Definition:] Minimally conforming processors must completely and correctly implement the ·Schema Component Constraints,· Validation Rules, and· Schema Information Set Contributions contained in this specification.*

Here, "Schema Information Set Contributions" (that is, additions to the document's infoset) are defined as follows (*XS1R*, 2.3):

> *[Definition:] Augmentations to post-schema-validation infosets expressed by schema components, which follow as a consequence of· validation and/or ·assessment· . Located in the fifth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Contributions to the post-schema-validation infoset (§C.2).*

The key words in this definition are *validation* and *assessment*, defined in the very beginning of *XS1R*, in Section 2 "Conceptual Framework," subsection 2.1 "Overview of XML Schema." We quote almost the entire subsection, highlighting key phrases:

> *An XML Schema consists of components such as type definitions and element declarations. These can be used to assess the validity of well-formed element and attribute information items . . . and furthermore may specify augmentations to those items and their descendants. This augmentation makes explicit information which may have been implicit in the original document, such as* ***normalized and/or default values for attributes and elements and the types of element and attribute information items****.*
>
> *Schema-validity assessment has two aspects:*
>
> *1. Determining local schema-validity, that is whether an element or attribute information item satisfies the constraints embodied in the relevant components of an XML Schema;*
>
> *2. Synthesizing an overall validation outcome for the item,* ***combining local schema-validity with the results of schema-validity assessments of its descendants, if any, and adding appropriate augmentations to the infoset to record this outcome*** *. . .*
>
> *Throughout this specification, [Definition:] the word assessment is used to refer to the overall process of local validation, schema-validity assessment and infoset augmentation.*

"Local validity assessment" is what RELAX NG does. DTDs, in addition, modify the infoset by adding default values, normalizing whitespace in some cases, and adding ID/IDREF constraints. XS goes beyond DTD, adding complex data type information. This approach is motivated by a certain vision of the Web as consisting of pipelines of XML applications that relegate all data validation tasks to the XS processor rather than performing them locally, within the application, based on the needs of the application. Not surprisingly, many developers have objected that they don't want structure validation bundled together with massive additions to the document's infoset; in particular, they want to have greater control over the kind of data validation that their applications use. A discussion on the xml-dev list in August 2001 expressed this point of view very clearly; see, for example, `http://lists.xml.org/archives/xml-dev/200108/msg01127.html`, `msg01131.html`, and `msg01136.html`.

There were calls for big changes in *XS1R*, if not abandoning it altogether. For whatever reasons (we think they were practical as well as political: *XSR1* was hugely delayed, holding up other recommendations, including *XSR2*, and *XSR2*, in turn, was needed for SOAP and Web services), XS1 was released as described. Fortunately, nobody (to the best of our knowledge) has seen a PSVI yet: implementers are too busy with all the details of "local validation."

A Simple Schema and a Validation Framework

In this section, we present an XS1 schema for Listing 8-13, and show how to run a validator. The schema is divided over two listings: Listing 8-14 shows most of the code, leaving a gap for the declaration of the type of `c` element, which is shown in Listing 8-15.

The schema (xsEx1nns.xsd) assumes that the document is in "no namespace." (That's what `nns` stands for.) For documents with namespaces, additional markup will be required as part of the mechanism by which the schema and the document find each other.

Listing 8-14. Our First XS1 Schema, xsEx1nns.xsd

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
<!-- An XS schema is an XML document in the XS namespace.
In this schema, XS is the default namespace; XS vocabulary is prefix-less -->
  <element name="r"><!-- start definition of r -->
    <complexType><!-- start definition of r type -->
     <sequence>
     <!-- the complex type of r is "zero or more repetitions of c" -->
      <element name="c" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <!-- see Listing 8-15 for what goes here -->
        </complexType>
      </element><!-- end of definition of c element -->
    </complexType><!-- end definition of r type -->
  </element><!-- end definition of r -->
</schema>
```

The type definition of `c` is in Listing 8-15. Note that we do not specify `minOccurs` and `maxOccurs` values for children of `c` because both have the default value of 1. (XS1 allows attribute defaults both in instance documents and in schemas themselves.)

Listing 8-15. Type Definition of the Type of Element `c`

```
         <sequence>
            <element name="gcStr" type="string" />
            <!-- minOccurs="1" maxOccurs="1" are defaults -->
            <element name="gcStrPat">
              <simpleType>
                <restriction base="string">
                  <pattern value="\d{2}-[A-Za-z]{3}-\w\d{2}"/>
                </restriction>
              </simpleType>
            </element>
            <element name="gcNum" type="decimal" />
<!-- XS2 defines built-in type 'year' but Xerces 1.4.1 does not support it,
       we use 'positiveInteger instead -->
            <element name="gcYear" type="positiveInteger" />
          </sequence>
```

With a document and a schema in place, we can try to validate the document. Before we do that, however, we have to make an addition to it: we have to specify what schema it wants to be validated against. RELAX NG, as you recall, relegates this question to those users of the data who want the data validated, allowing them to select the validating grammar. XS1 introduces specific markup in the instance document (and in the schema, for documents with namespaces, as we will see shortly).

Schema-Related Markup in the Document

A reference to the schema appears as an attribute of the document's root element. The attribute is a global attribute in the "Schema Instance" namespace, as seen in Listing 8-16.

Listing 8-16. Schema-Related Markup in a No-Namespace Document

```
<?xml version="1.0"?>
<r
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="xsEx1nns.xsd"
>
  <!-- the rest as in Listing 8-13 -->
</r>
```

Obviously, you may frequently want to validate an instance document against a schema of your own. (What if you don't trust the document?) For this reason, `xsi:noNamespaceSchemaLocation` and `xsi:schemaLocation` (coming up) are, according to XS1, only "hints" that the processors don't have to support and applications don't have to follow. In practice, all processors so far support them, but they also allow you to turn "automatic" schema validation off and program your own validation. The way this is done in Xerces is described in `http://xml.apache.org/xerces-j/properties.html`. We provide this option in our Web application that validates against XS schemas.

Framework for Validation

Our XS schema validation is set up in much the same way as with RELAX NG. The code is a little simpler because the validation is done by the parser (Xerces) itself, rather than by an external library. No additional libraries are needed; we just have to set the parser's properties for schema validation.

To connect to the application, refer to `http://localhost:8080/xmlp/xs/validate.htm`. You will see a page with two frames. The frame on the left has a form with several input elements. Submitting the form runs the validator. If the text area on the right shows no output, validation has been successful; otherwise, an error message is displayed. (See Figure 8-2.)

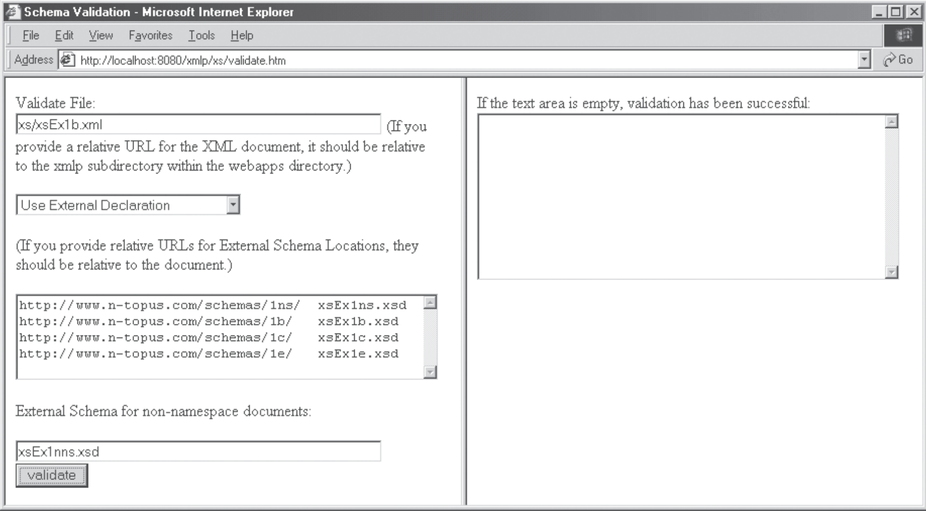

Figure 8-2. XML schema validation page

The form in the left frame has the following input elements:

- An input box for the XML document to validate.
- A selection element to indicate whether the document is to be validated against the schema it specifies or against an external schema of your choosing.
- A text area to indicate which namespace is to be validated by which schema. As you will see in a moment, a schema validates a vocabulary in a specific namespace, or a vocabulary that is not in any namespace.
- A last input element to specify an external schema to validate a document that is not in any namespace.

The form input, when submitted, goes to validate.jsp. In outline, validate.jsp is identical to validaterng.jsp. (See Listing 8-2 through 8-4.)

1. Import the libraries.
2. Declare methods for getting an XML reader and an error handler.
3. Process the Request object.
4. Set up output streams.
5. Parse, validate, and display errors (if any) in the text area.

Skipping the imports, we proceed directly to declarations. Obtaining the parser, as we said, is a little simpler than in RELAX NG: we only need a method, not a method within a class. The code for obtaining an error handler is unchanged from validaterng.jsp. We show both methods together in Listing 8-17.

Listing 8-17. XML Reader and Error Handler

```
public XMLReader createXMLReader() throws SAXException {
  XMLReader xr;
  try{
   xr=SAXParserFactory.newInstance().newSAXParser().getXMLReader();
// set the parser features for Schema validation
   xr.setFeature("http://xml.org/sax/features/validation",true);
   xr.setFeature("http://xml.org/sax/features/namespaces",true);
   xr.setFeature("http://apache.org/xml/features/validation/schema",true);
   xr.setFeature(
```

```
        "http://apache.org/xml/features/validation/schema-full-checking",true);
    }catch(Exception ex){throw new SAXException(ex);}
  return xr;
 }
 public ErrorHandler createErrorHandler(){
   return new DefaultHandler(){
     void printErr(SAXParseException e){
       System.err.println("[**] "+e+"\n    at "+e.getLineNumber()+":"
            +e.getColumnNumber()+" of "+e.getPublicId()+" - "+e.getSystemId() );
     }
     public void warning(SAXParseException e) throws SAXException{printErr(e);}
     public void error(SAXParseException e) throws SAXException{ printErr(e);}
     public void fatalError(SAXParseException e) throws SAXException{
       printErr(e);}
   };
 }
```

With declarations in place, we can proceed to the code that actually does the work. First, we set up output streams. (See Listing 8-18.) This part is unchanged from validaterng.jsp. (Remember that, in the Tomcat system, System.out and System.err are directed to the Tomcat console window; we want to direct error output to the text area in the Web page.)

Listing 8-18. Parsing and Schema Validation

```
java.io.PrintStream sysOut = System.out;
java.io.PrintStream sysErr = System.err;
java.io.ByteArrayOutputStream baos = new java.io.ByteArrayOutputStream();
java.io.PrintStream ps = new java.io.PrintStream(baos);
```

Finally, we process the Request object, parse (including schema validation), and display error messages, if any:

```
<html><body>If the text area is empty, validation has been successful:
<textarea cols="50" rows="10">
<% try{
// set up output streams, create XMLReader and its ErrorHandler
  System.setOut(ps);
  System.setErr(ps);
  XMLReader reader=createXMLReader();
  reader.setErrorHandler(createErrorHandler());
```

```
// process Request; set parser properties accordingly
  String uri= request.getParameter("uri");
   if(uri.indexOf(":") < 0)
     uri = new File(application.getRealPath(uri)).toURL().toString();
  String useExternalLoc=request.getParameter("external");
  String externalSchemaLocation=
      request.getParameter("externalSchemaLocation");
  String extNoNSLocation=
      request.getParameter("externalNoNSSchemaLocation");
  if("yes".equals(useExternalLoc)){
    if(externalSchemaLocation!=null&&externalSchemaLocation.length()>0)
         reader.setProperty(
          "http://apache.org/xml/properties/schema/external-schemaLocation",
           externalSchemaLocation);
    if(extNoNSLocation!=null&&extNoNSLocation.length()>0)
       reader.setProperty(
"http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation",
           extNoNSLocation);
    }
  reader.parse(uri); // the punch line
 }catch(Throwable ex){ex.printStackTrace(new java.io.PrintWriter(out,true));}
finally{System.setOut(sysOut);System.setErr(sysErr);}
%>
<%= baos.toString() %>
</textarea></body></html>
```

The rest of the chapter will use the validator of Listing 17 and 18.

What If the Document Is in a Namespace?

What if our simple document of Listing 13 and 16 has a markup vocabulary that is in a namespace? Changes would need to be introduced both in the document and in the schema.

In the document, obviously, we would have to declare the namespace. In addition, we would change the schema-instance attribute that references the schema. Instead of `noNamespaceSchemaLocation`, we now use `schemaLocation`. The value of `schemaLocation`, a quoted string, has internal structure: it has to consist of two tokens—the first of which is the document's namespace, the second the location of the schema. See Listing 8-19.

Listing 8-19. Schema-Related Markup in a Namespaced Document, xsEx1ns.xml

```
<?xml version="1.0"?>
<r xmlns="http://www.n-topus.com/schemas/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation ="http://www.n-topus.com/schemas/
                                    xsEx1a.xsd"
>
  <!-- the rest as in Listings 13 and 16 -->
</r>
```

The reason we repeat the document's schema in the schema-instance attribute is because we want to provide for the possibility that a single document contains vocabularies from several namespaces, each of which can be validated by a schema of its own. In other words, a schema validates a certain namespace. The value of `schemaLocation` can consist of any number of pairs that establish a connection between a namespace and a schema document that validates it. (There is no assumption that such a document has to be unique: the same namespace can be validated, in different ways, by different schemas.)

In the schema itself, the namespace it is supposed to validate is specified as the `targetNamespace` attribute, as shown in Listing 8-20.

Listing 8-20. Namespace-Related Markup in a Schema Document, xsEx1ns.xsd

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.n-topus.com/schemas/1ns/"
  elementFormDefault="qualified"
>
<!-- the rest copied unchanged from Listing 8-14 -->
  <element name="r"><!-- start definition of r -->
    <complexType><!-- start definition of r type -->
     <sequence>
     <!-- the complex type of r is "zero or more repetitions of c" -->
      <element name="c" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <!-- see Listing 8-15 for what goes here -->
        </complexType>
      </element><!-- end of definition of c element -->
    </complexType><!-- end definition of r type -->
  </element><!-- end definition of r -->
</schema>
```

The `targetNamespace` attribute is easy to understand, but what is `elementFormDefault`? (This particular item probably provoked more comment than any other single detail of the specification.) We will explain what it is about later in the chapter. For now just accept that it has to be there, and we don't recommend that you omit it from any schemas of your own.

Simple Schema Variations and Best Practices

We now return to the contents of the schema (as opposed to specifying how the document finds its schema). The schema of Listing 14 and 15 is organized as a single deeply nested structure. Costello's "Best Practices" calls this the *Russian Doll pattern*, in reference to the wood matryoshka dolls that nest inside each other. This section will present more modular alternatives to the Russian Doll.

Variation 1: Named Types

Reviewing Listing 8-14 and 8-15, you will notice that the XS `element` element has two alternative content models. One is to be empty, as in

```
<element name="gcNum" type="decimal" />
```

In this content model, the type of the element is declared as the value of the `type` attribute. The other content model is for `element` to have a child that is the definition of the element's type. Two examples from Listing 8-14 and 8-15 are

```
<element name="c" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <!-- see Listing 8-15 for what goes here -->
        </complexType>
</element><!-- end of definition of c element -->

<element name="gcStrPat">
        <simpleType>
          <restriction base="string">
            <pattern value="\d{2}-[A-Za-z]{3}-\w\d{2}"/>
          </restriction>
        </simpleType>
</element>
```

The type can be a simple type (`simpleType`) or a complex type (`complexType`), but we will disregard this difference for the moment. Our point of interest is that, if a type has a name, it can be the value of the `type` attribute. Built-in types, such

as decimal, of course have names, but XS1 allows the user to give names to user-defined types and use these names in element definitions. We are going to rewrite Listing 8-14 and 8-15 in that fashion. In particular, for each element that has something other than a built-in simple type, we will define a named type and use it in the declaration of the element.

More Namespaces

Our current intent is to define some names of types and to use those names elsewhere in the schema. Once you start using names, you have to ask yourself what namespace they are coming from. (Note that, in Listing 8-20, the schema identifies its target namespace and declares its vocabulary, but it does not itself use that vocabulary. For this reason, the body of the schema could remain unchanged from Listing 8-14 and 8-15.) The names we are going to define are internal to the schema: they do not appear at all in the document instance. However, XS1 stipulates that those names must be added to the target namespace. So, in our schema, we must not only specify what the target namespace is, but we must also declare it as a namespace that is used in the current document, the schema itself.

To make the vocabularies clearly visible, we will map both the XS namespace and the target namespace to a prefix, as shown in Listing 8-21.

Listing 8-21. Listing 8-14 and 8-15 Rewritten with Named Types (xsEx1b.xsd)

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.n-topus.com/schemas/1b/"
  xmlns:p="http://www.n-topus.com/schemas/1b/"
  elementFormDefault="qualified"
>
  <xs:element name="r" type="p:rType" />
  <xs:complexType name="rType">
   <xs:sequence>
    <xs:element name="c" minOccurs="0" maxOccurs="unbounded" type="p:cType" />
    </xs:sequence>
   </xs:complexType><!-- end definition of rType -->

  <xs:complexType name="cType">
   <xs:sequence>
     <xs:element name="gcStr" type="xs:string" />
     <xs:element name="gcStrPat" type="p:strPatType" />
     <xs:element name="gcNum" type="xs:decimal" />
     <xs:element name="gcYear" type="xs:positiveInteger" />
```

```
    </xs:sequence>
  </xs:complexType><!-- end definition of cType -->
  <xs:simpleType name="strPatType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{2}-[A-Za-z]{3}-\w\d{2}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

We provide a separate document, xsEx1b.xml, that is validated by this schema. Its only difference from xsEx1ns.xml is that it references the schema of Listing 8-21 rather than the schema of Listing 8-20.

To recapitulate the main point, Listing 8-21 has two namespaces for two vocabularies. One is the schema vocabulary (which includes both the “structure-definition” vocabulary such as `element`, `attribute`, and `complexType`, and the “simple-types” vocabulary, such as `string`, `decimal`, and `positiveInteger`). The other is the target vocabulary of element and attribute names defined by the schema. This vocabulary of the target namespace also includes all the type names that are defined in our schema, even though they never appear in instance documents. If target namespace vocabulary is used in the schema itself, the target namespace must be declared, and the vocabulary items must use qualified names.

Listing 8-21 declares both element names and type names, but it uses only the type names. Element names are not used because all elements are declared where they belong in the instance structure. For instance, the element `gcString` is a child of the element `c`, and its declaration is the child of the declaration of `c`. In our next variation, we will declare all elements at the top level, so their declarations are all children of the `schema` element. In defining content models, we will refer to those top-level declarations. In the terminology of XS1, element declarations that are children of schema are called *global declarations*, whereas declarations embedded within other declarations are called *local*. Our next version of Listing 8-15 will have all declarations global.

Global vs. Local Declarations and Element References

As before, we will map both the schema namespace and the target namespace to a visible prefix, to see the vocabularies clearly. The schema is in Listing 8-22 (xsEx1c.xsd).

Listing 8-22. Listing 21 Rewritten with Global Declarations

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.n-topus.com/schemas/1c/"
   xmlns:p="http://www.n-topus.com/schemas/"
>
  <xs:element name="r" type="p:rType" />
  <xs:complexType name="rType">
   <xs:sequence>
    <xs:element ref="p:c" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType><!-- end definition of rType -->
  <xs:element name="c" type="p:cType" />
  <xs:complexType name="cType">
   <xs:sequence>
     <xs:element ref="p:gcStr" />
     <xs:element ref="p:gcStrPat" />
     <xs:element ref="p:gcNum" />
     <xs:element ref="p:gcYear" />
    </xs:sequence>
  </xs:complexType><!-- end definition of cType -->
  <xs:element name="gcStr" type="xs:string" />
  <xs:element name="gcStrPat" type="p:strPatType" />
  <xs:element name="gcNum" type="xs:decimal" />
  <xs:element name="gcYear" type="xs:positiveInteger" />
  <xs:simpleType name="strPatType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{2}-[A-Za-z]{3}-\w\d{2}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

In this schema, everything is given a global name: user-defined types and elements that are children of some other elements and could be declared "locally," within the declarations of their parent elements. In terms of style, Listing 8-22 is the complete opposite of the Russian Doll style of Listings 8-14 and 8-15. Which style is better? This is the subject of the "Best Practices" discussions.

Choices and Best Practices

The author of an XS schema faces the following choices, among others:

- Make the target namespace or the XS namespace (or neither) the default.
- Declare all elements globally so that the declarations are all children of schema, or declare elements that are "local" to another element locally, within the declaration of that other element.
- Use named types or anonymous types.

These and similar choices are the subject matter of frequent "Best Practices" discussions on the xml-dev list and elsewhere. Roger Costello's xsBest page at `http://www.xfront.com/BestPracticesHomepage.html` is a central point where these discussions are collected and summarized in clear language and uniform format.

Global, Local, and the `elementFormDefault` *Attribute*

Yet another choice that has been much discussed on xsBest and elsewhere concerns the `elementFormDefault` attribute. If you look back at the `schema` element of Listing 8-22, you will notice that the attribute is missing: the schema validates without it. The reason is that this attribute is relevant only when there are local declarations. Its possible values are `qualified` and `unqualified`, and the choice they regulate is this: do locally declared element names belong in the target namespace or in no namespace? In other words, with the schemas of Ex1ns.xsd and Ex1b.xsd, if the value of the attribute is `qualified`, then the document of Listing 8-23 is valid; if the value of the attribute is `unqualified`, then the document of Listing 8-24 is valid. (Listing 8-23 is xsEx1d.xml, validated by Ex1ns.xsd and Ex1b.xsd. Listing 8-24 is xsEx1e.xml, validated by Ex1e.xsd.)

Listing 8-23. Document-Matching Schema with `elementFormDefault=qualified`

```
<q:r xmlns:q="http://www.n-topus.com/schemas/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation ="http://www.n-topus.com/schemas/
                           xsEx1b.xsd"
>
  <q:c>
    <q:gcStr>any string</q:gcStr>
    <q:gcStrPat>22-abc-z12</q:gcStrPat><!-- a string pattern -->
    <q:gcNum>123.45</q:gcNum><!-- a number -->
```

```
    <q:gcYear>1999</q:gcYear><!-- a year -->
  </q:c>
  <!-- many more children -->
</q:r>
```

Listing 8-24. Document-Matching Schema with `elementFormDefault=unqualified`

```
<q:r xmlns:q="http://www.n-topus.com/schemas/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation ="http://www.n-topus.com/schemas/
                          xsEx1b.xsd"
>
  <c>
    <gcStr>any string</gcStr>
    <gcStrPat>22-abc-z12</gcStrPat><!-- a string pattern -->
    <gcNum>123.45</gcNum><!-- a number -->
    <gcYear>1999</gcYear><!-- a year -->
  </c>
  <!-- many more children -->
</q:r>
```

Somewhat surprisingly, the value `unqualified` is the default: if you omit it in the schemas of Listing 8-20 and 8-21, the document of Listing 8-24 will check out but the document of Listing 8-23 will bomb. The reasoning seems to be as follows: elements declared locally are part of the content model of some other elements, and therefore their names are not exposed and need not be protected. If we think of XS1 as a data modeling language, the reasoning makes sense: local names would unnecessarily clutter the namespace, obscuring the main divisions in the data. However, this is a big *if*: why would you want the same language to perform both syntactic validation and data modeling? One could argue, and some people have, that it would be better to use a dedicated language like UML for data modeling and provide a UML–XML mapping.

XS1's ambition to be a modeling language extends far beyond the `elementFormDefault` attribute: the types of XS1 and XS2 form an inheritance hierarchy with a single root in `anyType`. User-defined types are all derived from that type and can themselves stand in the base-type/derived-type relationship. Types can be *abstract*, forcing definition of derived types, or they can be *final*, preventing definition of derived types. Whereas simple types can be derived only by restriction, complex types can be derived either by restriction or by extension.

Just as in object-oriented programming, you can substitute a base type for a derived type, even if the derived type was formed by restriction. You can block such substitution using the `block` attribute. You can also establish substitution relation on element names, using the `substitutionGroup` element: if type D is the same as or is derived from some base type B, then you can specify that globally

defined elements of type D—whatever their name—can be substituted into content models that require an element of type B. For instance, if Lager and Ale are declared globally and have schema types that are derived from the BeerType, then we specify that we don't mind having a Lager wherever a Beer is required by the grammar:

```
<xs:element name="Beer" type="xs:BeerType"/>
<xs:element name="Lager" substitutionGroup="Beer" type="xs:LagerType"/>
<xs:element name="Ale" substitutionGroup="Beer" type="xs:AleType"/>
<xs:element name="Porter" substitutionGroup="Beer" type="xs:PorterType"/>
<xs:element name="Stout" substitutionGroup="Beer" type="xs:Stout"/>
```

It would go well beyond the scope of this chapter or this book to illustrate and discuss all these possibilities. We will provide a concise overview in the next section, and otherwise recommend Roger Costello's excellent tutorial and many other online sources.

XS1 Overview

This is a cut-and-dry summary of many pages of dense documentation. It has many bulleted lists, and very few examples.

We use the conventions of XS1, illustrated by this example:

```
elementFormDefault = (qualified | unqualified) : unqualified
```

This means "the attribute `elementFormDefault` can have two possible values, `qualified` or `unqualified`, with `unqualified` being the default."

What Is There?

This is a complete list of things you can find in an XS schema. Items in parentheses are details that we are going to say very little or nothing about.

- annotations
- simple type definitions
- complex type definitions
- element declarations

- attribute declarations
- (particles, wildcards, attribute uses)
- (attribute group definitions and attribute groups)
- (model group definitions and model groups)
- (identity-constraint definitions: a much more powerful version of ID/IDREF)
- (notation declarations: virtually never used)

Annotations, Extending Schemas

Almost any XS element, including the root `schema` element, can have an `annotation` element as its child, sometimes specifically as its first child. The `annotation` element can contain either human-readable or machine-processable materials; the latter can express additional constraints that are not expressible in XS. A good use of `annotation` elements is to place Schematron constraints in them. (On Schematron and how to use it to extend schemas with additional constraints, see Costello's `http://www.xfront.com/ExtendingSchemas.pdf`.)

The Root Element

The root `schema` element can have the following attributes:

- targetNamespace = anyURI
- `elementFormDefault` = (qualified | unqualified) : unqualified
- `attributeFormDefault` = (qualified | unqualified) : unqualified (Same idea as for `elementFormDefault`)
- `id` = ID
- `version` = token
- `xml:lang` = language

- `blockDefault` = (#all | List of (extension | restriction | substitution)) (The default value for `block` attributes that block substitution of base types for derived types or elements for their declared substitutions. You can block them all or be more specific.)

- `finalDefault` = (#all | List of (extension | restriction)) (the default value for `final` attributes that block derivation, by extension or restriction or both)

The content model of the `schema` element is

```
((include | import | redefine | annotation)*,
(
  ((simpleType | complexType | group | attributeGroup) |
   element | attribute | notation), annotation*)*
)
```

This can be summarized as

1. First, do all your `include`'s, `import`'s, `redefine`'s, and top-level annotations.

2. Then do all your types, groups, elements, and attributes, in any order, sprinkling with annotations liberally.

3. Ignore notations, nobody uses them.

`include`, `import`, *and* `redefine`

The `include` element is to include another schema with the same target namespace. The `import` element is to include another schema with a different target namespace. This is useful when your document uses more than one namespace.

Both `include` and `import` have a `schemaLocation` attribute. In addition, `import` has a `namespace` attribute.

The `redefine` element does the same as `include` and additionally redefines one or more definitions in the included schema. It has a `schemaLocation` attribute. Its content are (re)definitions of `simpleType`, `complexType`, `attributeGroup`, or `group`.

`group, attributeGroup, modelGroup`

You can group several element declarations into a group and give such a group a name, for later reuse. Similarly, you can group several attribute declarations into a group and give such a group a name for later reuse. You have to use the `group` element to group element declarations and the `attributeGroup` element to group attribute declarations. Finally, you can group pieces of a content model into a group (using the `modelGroup` element) and reuse that in definitions of other content models.

Declaring Elements

There are three patterns for declaring elements, shown in Listing 8-25:

- with a named type, specified by the `type` attribute
- with an unnamed complex type
- with an unnamed simple type

Listing 8-25. Declaring Elements

```
<xs:element name="name" type="type" minOccurs="int" maxOccurs="int"/>
<xs:element name="name" minOccurs="int" maxOccurs="int">
       <xs:complexType>
      ...
       </xs:complexType>
</xs:element>
<xs:element name="name" minOccurs="int" maxOccurs="int">
       <xs:simpleType>
              <xs:restriction base="type">
             ...
              </xs:restriction>
       </xs:simpleType>
</xs:element>
```

> **NOTE** *An element declaration may contain a* `default="value"` *attribute, providing a default value for the element. In the document, such elements are entered with empty content.*

Declaring Attributes

There are two patterns for declaring attributes:

- with a named type, specified by the `type` attribute
- with an unnamed simple type

```
<xs:attribute name="name" type="simple-type" use="how-used"
                        default/fixed="value"/>
<xs:attribute name="name" use="how-used" default/fixed="value" >
       <xs:simpleType>
              <xs:restriction base="simple-type">
                   ...
              </xs:restriction>
       </xs:simpleType>
</xs:attribute>
```

The `use` attribute has these possible values: `required`, `optional`, or `prohibited`. If there is a `default` or a `fixed` attribute, there must be no `use` attribute. The value of a `default` or `fixed` attribute is a simple type value.

In content model (complex type) definitions, attribute declarations, for some reason, must come last, after all the element declarations.

Defining Complex Types

There are four patterns for defining complex types.

- complex type that is not derived from another user-defined type or from a simple type (that is, complex type derived directly from `anyType`).
- complex type that is derived from another user-defined complex type by extension
- complex type that is derived from another user-defined complex type by restriction
- complex type that is derived from a simple type (built-in or user-defined) by adding attributes

The definitions can be global (with a `name` attribute) or embedded (without a `name` attribute).

In the patterns shown in Listing 8-26, whenever we say `sequence`, we actually mean any one of `sequence`, `choice`, `group`, or `all`. The `all` element means "a sequence in any order" (same as `interleave` of RELAX NG).

Listing 8-26. Declaring Complex Types

```
<xs:complexType>
    <xs:sequence> <!-- can be sequence or choice or group or all -->
   ...
    </xs:sequence>
</xs:complexType>

<xs:complexType>
  <xs:complexContent>
     <xs:extension base="complex type" >
        <xs:sequence>
       ...
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType>
  <xs:complexContent>
     <xs:restriction base="complex type" >
        <xs:sequence>
       ...
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType>
  <xs:simpleContent>
     <xs:extension base="simple type" >
         <xs:attribute name="name" type="simple type".../>
        ...
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

NOTE *The empty element with attributes is treated as a complex type.*

Wildcards for Extensibility

XS1 provides `any` and `anyAttribute` wildcards that match any element and attribute (respectively) in the instance document. Both can be constrained by a namespace attribute to mean "any element/attribute from the specified namespace, or from the target namespace, or from any namespace other than the target namespace, or from no namespace at all":

```
<xs:any.../>
<xs:any namespace="http://www.w3.org/1999/XSL/Transform"/>
<xs:any namespace="##targetNamespace"/>
<xs:any namespace="##other".../>
<any namespace="##local"/><!-- any element in no namespace -->
<xs:anyAttribute namespace="http://www.w3.org/XML/1998/namespace"/>
```

For instance, the following declaration allows any attribute to be added to the element:

```
<xsd:element name="ExtendMeWithAttribute">
  <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
               <xsd:anyAttribute/>
            </xsd:extension>
      </xsd:simpleContent>
  </xsd:complexType>
<xsd:/element>
```

Schema-Related Markup in Document Instances

Schema-related markup in document instances is introduced by global attributes in the schema-instance namespace. (As a notational convenience, we will assume that the namespace has been mapped to the `xsi:` prefix.) You have seen two such attributes: `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation`. There are two more: `xsi:nil="true"` and `xsi:type`.

`xsi:nil="true"` in an instant document's element means that the element is empty even though it is not declared as empty. The schema declaration of the element must have the `nillable` attribute set to true. This is clearly added for database uses, and it invites the familiar kind of criticism: why cram this into a specification that is intended to be used by everybody, including people who never use databases?

`xsi:type` is a brute-force mechanism to specify the type of an element or an attribute in an instance document bypassing the entire schema validation and PSVI machinery. The value of the attribute can be anything that makes sense to the receiving application; typically, this includes all XS2 simple types. SOAP documents and applications use this attribute a lot, as you will see in the next chapter.

Conclusion

In this chapter, we covered RELAX NG and XML Schema, two grammar formalisms that are intended as an improvement upon—and a replacement for—the DTD. In particular, both RELAX NG and XML Schema

- use XML syntax,
- support XML namespaces,
- support unordered content,
- integrate attributes into content models, and
- support data typing.

RELAX NG has excellent support for modularization and reuse, which makes its implementation of XHTML modularization and XHTML Basic very easy to use. We have looked at the implementation closely and used it to define a RELAX NG grammar for RDDL.

RELAX NG support for data typing is layered on top of validation: the user can perform validation or data typing or both. RELAX NG does not itself provide a data type library but allows the grammar to specify such a library. The Jing implementation of RELAX NG comes with built-in support for the data type library of XS2. We have presented this library and provided examples of simple type derivations in XML Schema. Finally, we have shown how user-defined types are handled in RELAX NG.

XS1 is a very large specification with many powerful features. We have tried to do it justice to the extent that it is possible in a single section. It has many powerful features, and its many critics suggest that it should be refactored into several smaller modules, each of which are useable in different contexts.

CHAPTER 9

Web Services

IN THIS CHAPTER, we will present Web services, probably the most active and rapidly growing area of XML-based applications. We will explain what Web services are and what they are not, we will introduce the main specifications that define Web services functionality, and we will work through two complete examples that illustrate how Web services are built and deployed. In outline, the chapter proceeds as follows:

- What's a Web service?
- a simple example
- client and server variations
- SOAP in detail
- a less simple example
- publishing a business with UDDI

As our framework, we will be using Apache Axis, but some examples will include JScript MSXML, or standard ECMAScript code in the browser and JSPs that don't use any Axis-specific classes.

What's a Web Service?

No official definition of *Web services* is enshrined in some sort of a standard or a W3C recommendation. However, there is a general agreement that a Web service is a distributed application that allows maximum interoperability between components written in different languages and running on different platforms. It is also generally agreed upon that this kind of interoperability is achieved by encoding interactions between components in an XML protocol.

In practice, the concept of a Web service (WS) is both more specific (there is a strong candidate for the position of the XML protocol) and more general, in

that WSs have a number of additional features. These features of a WS, illustrated by our examples, are as follows:

- It uses an agreed-upon, XML-based message format called *Simple Object Access Protocol* (*SOAP*). SOAP message may or may not be delivered over HTTP. (FTP, SMTP, and even proprietary messaging architectures are also a possibility.)

- It provides a meta-description of its access point(s) and interfaces in an XML-based Web Services Description Language (WSDL).

- It is registered with one of several synchronized, online registries that maintain their entries in an agreed-upon, XML-based format. The format is called *Universal Description, Discovery, and Invocation* (*UDDI*) because the purpose of the registries is to enable automatic service discovery and invocation.

The access point of a Web service and the Internet address of a UDDI registry are likely to be URLs. Otherwise, a Web service doesn't have to be of any service to anybody, and may have nothing to do with the Web. In particular, the user interface to a Web service doesn't have to be a Web browser (it can be, for example, a .NET Windows form instead, or, indeed, an Excel spreadsheet that invokes a native HTTP object), and the SOAP server that dispenses SOAP messages may be completely unrelated to any Web server.

The Vision

Figure 9-1 is a well-known diagram that neatly summarizes the Web services vision.

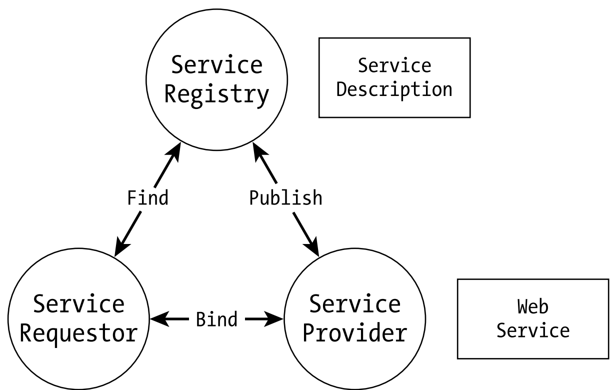

Figure 9-1. Publish, find, and bind

In this diagram, the "publish" step is the first and is performed once: a Web service publishes its registry description (which contains, among other things, its WSDL description) in a service registry (most likely, a UDDI registry). After that, a customer looking for a service of that type can find it in the registry and use the registry information to "bind" itself to the service. A productive exchange of services (and, in most cases, money) takes place.

We will return to this high-level diagram after we learn the mechanics of low-level SOAP exchanges.

Areas of Application

Two main areas of application are envisaged for Web services. One is automatic service discovery and invocation on the Internet by software agents that construct long chains of interactions without human intervention. The other is legacy data and application integration. Web services wrap all their data exchanges into a standard XML format that is understood by any other Web service running on any platform and implemented in any programming language. To bring legacy formats into a distributed application, one only has to write a Web service that encodes those formats as SOAP messages.

The first of these areas has received by far the most attention and is a frequent subject of media discussions and futuristic scenarios. (The word *hype*, which we mostly try to avoid, may be appropriate here.) Consider the standard example of a travel reservation service. Currently, a person finds a reservation service on the Web and uses it to connect to a database of some sort and view the options; once an option is selected, another process, again initiated by the human user, results in booking and online payment, and, after that, the same overworked user has to arrange for a hotel reservation and car rental. Web services are supposed to automate much of this: the user supplies the parameters of the situation to an intelligent agent of some sort that initiates a chain of interactions among several specialized services, such as airline booking, hotel booking, and car rental, based on the user profile and previous history.

Scenarios like this determine much of the content of SOAP and other specifications. This is not surprising: a specification has to be future oriented to avoid being obsolete on arrival. However, a careful analysis suggests that, in this case, the specifications may be aiming too far ahead. Futuristic scenarios of the kind we've just described are very distant because they involve two kinds of difficult issues. One is more technical: for software agents to cooperate, the problems of security, quality of service, payment, and enforceable contracts among Web services have to be resolved in some standard, interoperable ways. The corresponding specifications are barely on the drawing board. The other set of difficult issues is not even technical yet, but conceptual: cooperation among human agents is based on deep-shared context that several decades of artificial

intelligence have been unable to formalize. The notion that Web services will suddenly succeed where previous long-term efforts proved fruitless doesn't seem to be based on any solid grounds. We highly recommend two recent articles that discuss these issues in great detail:
`http://www.xml.com/pub/a/2002/02/06/webservices.html` and
`http://www.xml.com/lpt/a/2001/10/03/webservices.html`.

This said, the use of Web services in a "shared context" situation is likely to blossom. This particularly applies to legacy data and application integration within an enterprise, wherein the semantics of lexical items can be agreed upon, Web service interfaces can be kept stable and changed in sync, and problems of security and payment are either nonexistent or easy to solve.

Web Services and the Programmer

An essential part of the Web services vision is that creating a Web service must be easy for a programmer. Perhaps the main area of competition between providers of Web services infrastructure is in programming environments and tools for automatic code generation: generating SOAP servers from Java or C# classes, generating WSDL descriptions from SOAP servers, and generating SOAP client skeletons from WSDL descriptions. This is, of course, good news for programmers, as long as one of the providers doesn't clobber the rest into nonexistence or irrelevance and becomes a de facto monopolist.

In this chapter, we use the Web services infrastructure from Apache, called Axis. As of this writing (February 2002), it is in a third alpha release and relatively stable. (Be aware that the framework is under active development, and there are frequent nightly builds, often in response to recently reported bug or a feature request.) Other important infrastructures include Microsoft's .NET, IBM's WebSphere, and Sun's JAX-RPC (`http://java.sun.com/xml/jaxrpc/`).

Components of a Service

A service exposes at least one "endpoint," usually by publishing its URI. A consumer of the service and the service endpoint exchange SOAP messages. The consumer may be a passive receiver of messages broadcast by the service or it may initiate an exchange by sending a request and receiving a response. The elementary communication step is a single message, but messages may form more-complex Message Exchange Patterns (MEPs). The most common MEP—and the most common use of Web services—is a Remote Procedure Call (RPC): the consumer requests some computation and sends in parameters, and the service returns the results of the computation.

At the SOAP communication level, the service consumer is a "SOAP client," and the service itself is a "SOAP server." Both are computer programs. In the

examples of this chapter, SOAP clients are usually JSPs and SOAP servers are Java classes, but the client and the server don't have to be written in the same language or be running on the same platform because all their exchanges are XML encoded. It is this kind of interoperability that makes Web services so promising.

A SOAP server may or may not need a user interface, but a SOAP client usually does. The user interface may or may not be on the same computer as the SOAP client. In our examples, the user interface is a Web page with a form that sends information to the SOAP client (a JSP) over HTTP, but this is just one of many possible configurations. The SOAP specification is completely silent about how SOAP clients are accessed; its main concerns are the message exchange model, the structure of the message, and the "binding" of SOAP messages into an underlying protocol. HTTP is the only protocol that SOAP implementations must support, but other protocols can also be used, and examples of using email (SMTP) for exchanging SOAP messages are not difficult to find. In the situation of an internal network, with Web services used for legacy data integration, one can imagine SOAP messages traveling over a proprietary protocol.

Figure 9-2 summarizes the components of a Web service.

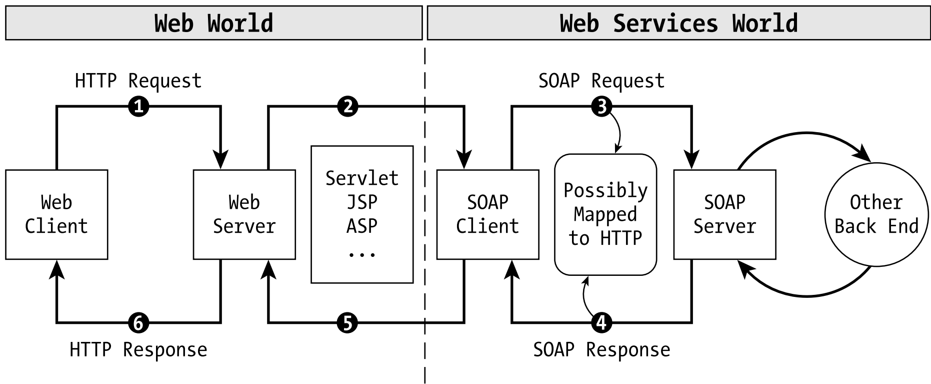

Figure 9-2. Components of a Web service

This diagram specifically shows the configuration of our first example. Although the overall structure of components is the same for all Web services, their implementation and binding into the underlying protocol can vary, as we have discussed.

Specification Map

This section summarizes the current status of the SOAP, WSDL, and UDDI specifications. We felt it was needed because the specifications are frequently referred

to as *standards* as in “Web services are based on the SOAP, WSDL, and UDDI standards.” In fact, they are not standards unless the term is stretched to the point of meaninglessness.

SOAP

The SOAP specification is a working draft (WD) at W3C and carries the usual WD caveats: “This is a public W3C Working Draft. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than ‘work in progress’” (`http://www.w3.org/TR/soap12-part1/`, “Status of this Document”). In fact, SOAP has gone through a major change in its transition from version 1.1 to 1.2, and many parts of the latest 1.2 version are clearly far from being stable.

SOAP version 1.2 comes in three parts: Part 0, nonnormative, is a primer; Part 1 is the “core” of the specification, and Part 2 is called “Adjuncts”. Part 1 describes the Message Exchange Model, the structure of a SOAP message, and the general principles of binding SOAP messages to the underlying protocol. Part 2 describes a specific binding (to HTTP), a specific SOAP encoding of data types as XML elements, and rules for using SOAP in an RPC framework. Until version 1.2, these three items were part of the same monolithic core, but, in an attempt to make the specification more modular, they were split off into Adjuncts. SOAP encoding, in particular, used to be a central part of the SOAP specification, prompting objections that individual developers and toolset vendors may want to develop their own procedures for serializing data objects to XML, and standardizing such procedures contributes nothing to Web service interoperability.

In SOAP messages, the encoding is specified by the `encodingStyle` attribute, whose value is a URI that identifies the encoding. (Just as with namespace URIs, an encoding style URI doesn’t have to point to an existing resource.) The `encodingStyle` URI for the encoding of SOAP 1.2 Part 2 is `http://schemas.xmlsoap.org/soap/encoding/`. (This URI actually points to an XS schema defining the encoding.) You will see that particular encoding used in our examples later in the chapter. Although no longer a normative part of the standard, it is very well designed, and a large body of installed software still uses it.

WSDL

Unlike SOAP, WSDL is just a W3C note (`http://www.w3.org/TR/wsdl`), with no commitment to ever making it into an area of activity and establishing a working group. The section entitled “Status” reads “This document is a NOTE made available by the W3C for discussion only. Publication of this Note by W3C indicates no endorsement by W3C or the W3C Team, or any W3C Members. W3C has had no

editorial control over the preparation of this Note." This status has not changed since March 2001 when version 1.1 of the specification was submitted by Microsoft, IBM, and Ariba. In December 2001, two of the editors of SOAP 1.2 published an article on `XML.com` entitled "All We Want for Christmas is a WSDL Working Group" (`http://www.xml.com/pub/a/2001/12/19/wsdlwg.html`). Their wish was granted in January 2002 when W3C formed a Web services activity that contains three working groups, including a Web services description WG. However, WSDL is not mentioned as a relevant document for the WG to consider.

In the meantime, in February 2002, yet another vendor consortium appeared, named *Web Services Interoperability Organization* (*WSIO*); one of its objectives is to "encourage best-practices use of 'baseline' Web services today (XML, SOAP, WSDL, UDDI)." (See `http://www.ws-i.org/FAQ.aspx#A04`.) Its members include most major software companies, such as IBM, Microsoft, Oracle, and SAP; Sun has not joined but may do so in the future. Neither WSIO nor the W3C working group has published any documents yet (as of February 2002), but they will probably work in consultation because many companies are represented in both organizations. It seems reasonably certain that something like WSDL will be standardized within a year, but it may bear little or no resemblance to WSDL. In the meantime, WSDL is deeply embedded in all the existing toolsets for developing Web services, as you will see in a moment.

UDDI

UDDI is even less of a standard than WSDL: it has never been submitted to W3C or any standards organization. It does have the backing of a powerful consortium, `uddi.org`, and it has toolset support. However, there are alternative approaches to publishing Web services, such as Web Services Inspection Language (WSIL) from IBM and Microsoft. (See `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`.) WSIL is easier to set up and use (in particular, it does not require a database), but UDDI offers more features, including taxonomic categorization of services and searches based on those taxonomies. IBM's Web Services Tool Kit (WSTK) 3.0 supports both. You can download the free 46MB archive from
`http://www.alphaworks.ibm.com/tech/webservicestoolkit`.

An Example

Our Web service will implement a remote procedure call. The procedure will receive one parameter, an integer, and return the prime factors of that integer. We will write two versions of the service: one will return prime factors as a string, with integers separated by the asterisk character (*), and the other will return

prime factors as an array of integers. An invocation of the array version is shown in Figure 9-3.

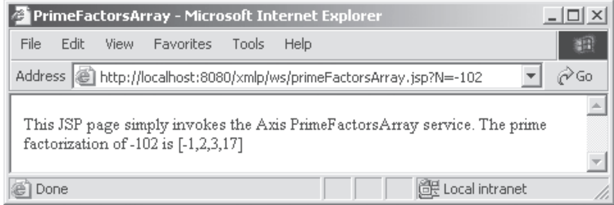

Figure 9-3. Web service RPC

We will start with the simpler string version, proceeding in the following order:

1. Define a Java class with a method that implements the Web service functionality.
2. Convert the Java class into a SOAP server.
3. Use either the SOAP server code or the initial Java code to generate a WSDL description of the service.
4. Use the Wsdl2Java utility to generate a SOAP client from the WSDL description. In addition to the client itself, the utility will generate a SOAP server "stub": a local class that exposes the same interfaces as the server and functions as a local proxy for the server.
5. Write a JSP that will invoke the SOAP client.

Because our service receives a single integer parameter, we will not bother with an HTML page to submit that parameter to the JSP; we will simply attach a query string, such as `?N=-102`, to the JSP's URL, as in Figure 9-3.

All Web service components, as implemented in our example, will live on a single computer that will also run the Web browser to connect to the JSP. Because that single computer will act as three computers on a network, Figure 9-4 provides a map (basically, Figure 9-2 with additional detail) showing where each component resides.

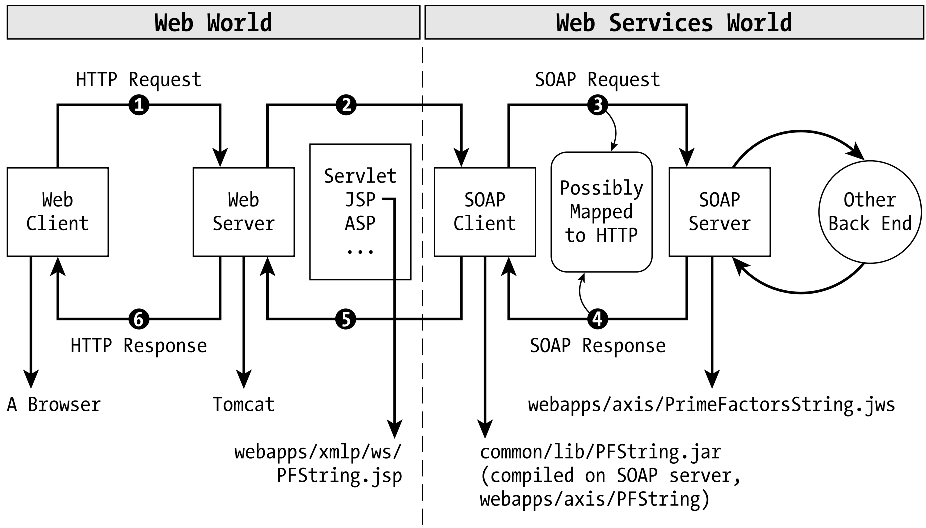

Figure 9-4. Components of a service, with their places of residence

We will implement the components of this diagram moving from right to left: SOAP server, SOAP client, and JSP. In the process, we will write a few useful macros and show you how to use TCPMon, a tool that allows you to inspect HTTP messages going to and from a web server. With TCPMon, you can view the actual SOAP messages that get exchanged between the (SOAP) client and server. At this point, we will be ready to discuss the structure of a SOAP message in detail.

The Java Class for a Web Service

The Java class has a single public method, `factors()`, that will become the operation supported by the Web service. (See Listing 9-1.)

Listing 9-1. The PrimeFactorsString Class

```
public class PrimeFactorsString {
/*
    factors(int N) produces a string result, as a String:
    factors(0) == "0"
    factors(-N) == "-" + factors(N)
    factors(12) == "2*2*3"
*/
public String factors (int N){
  if(N==0) return "0";
  if(N<0) return "-"+factors(-N);
  StringBuffer sB=new StringBuffer();
```

```
  int limit = (int)Math.floor(Math.sqrt(N));
  for(int i=2;i<=limit;i++)
    while(0==N%i) {
      sB.append(i);
      sB.append("*");
      N/=i;
    } // end of while loop and entire for loop
    if(sB.length() > 0) // the last "*" is not needed
    sB.deleteCharAt(sB.length()-1);
    return sB.toString();
  } // end of factors()
}  // end of class definition
```

From Java Class to a SOAP Server

Every Web service framework provides facilities for automatic conversion from application code to Web service code. Axis provides two methods: one simple but limited in its scope, and the other more involved and more flexible. The simple method is very simple indeed. It consists of two steps:

1. Rename your *.java file as a *.jws file.

2. Put it in the right place.

In our example, we rename PrimeFactorString.java as PrimeFactorString.jws and put it into the TOMCAT_HOME/webapps/axis directory. (You will find it there when you unzip our archive.) The SOAP server is ready to receive requests from a SOAP client. When a request comes in, Axis automatically locates the file, compiles the class, and converts SOAP messages into Java invocations of your service class.

The method has limitations. For one thing, it depends on access to source code. What if you want to make a compiled class into a Web service? Even more significantly, it does not provide any customization "hooks," such as user-defined classes in SOAP server code that are mapped to and from XML elements in SOAP messages. For a more flexible way of creating a Web service, we use a "deployment descriptor" written in Web Services Deployment Descriptor language (WSDD). We will show an example of WSDD use in our example of SOAP exchanges that involve structured objects and arrays.

Generating WSDL Descriptions

To generate (the first time around) and view the WSDL description of our Web service, point your browser at `http://localhost:8080/axis/PrimeFactorsString.jws?WSDL`. You will see a pretty verbose (some call it "bloated") WSDL file, which we show (Listing 9-2) slightly shortened and with important elements highlighted.

Listing 9-2. WSDL for PrimeFactorsString Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace=
    "http://localhost:8080/axis/PrimeFactorsString.jws"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:serviceNS="http://localhost:8080/axis/PrimeFactorsString.jws"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="factorsRequest">
    <part name="arg0" type="xsd:int"/>
  </message>
  <message name="factorsResponse">
    <part name="factorsResult" type="xsd:string"/>
  </message>
  <portType name="PrimeFactorsStringPortType">
    <operation name="factors">
      <input message="serviceNS:factorsRequest"/>
      <output message="serviceNS:factorsResponse"/>
    </operation>
  </portType>

  <binding
      name="PrimeFactorsStringSoapBinding"
      type="serviceNS:PrimeFactorsStringPortType">
    <soap:binding
        style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="factors">
      <soap:operation soapAction="" style="rpc"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace=""
          use="encoded"/>
      </input>
```

```
        <output>
          <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace=""
            use="encoded"/>
        </output>
      </operation>
  </binding>
  <service name="PrimeFactorsString">
      <port
        binding="serviceNS:PrimeFactorsStringSoapBinding"
        name="PrimeFactorsStringPort">
        <soap:address
         location="http://localhost:8080/axis/PrimeFactorsString.jws"/>
      </port>
  </service>
</definitions>
```

It makes little sense to discuss WSDL in detail because it is likely to undergo changes before the book is out, and it will certainly change a great deal before it becomes any kind of standard. For now, just note these main structural points:

- The file begins by specifying all the messages used by the service (just two in our example).
- Each message lists its parameters and their data types; the data types are those of XML Schema Part 2 (`xsd:int` and `xsd:string` in our example).
- The `portType` is defined in terms of it operations (only one, `factors`, in our case), and the operations are defined in terms of messages they send and receive.
- The `binding` gives the SOAP specifics (message pattern is RPC and the underlying protocol is HTTP), and restates the operations in more SOAP-y terms. The SOAP encoding is specified by a URI; the "target" namespace of service-specific elements within the SOAP message is set to the empty string. When we finally get to SOAP messages received and sent by this service, you will see how these settings determine the composition of the message.
- The final part, `service`, specifies the port and the URL for SOAP messages to use.

In theory, a programmer doesn't have to know much about WSDL or ever look at WSDL files. In practice, as usual with generated code, some post-editing will probably be needed in nontrivial applications.

Our next question is: what is it good for? The answer is: for automatic code generation, either within the same framework, or an alternative Java framework, or indeed in any framework that has a WSDL-to-code transformer, which includes all of those that are worth considering. An essential feature of Web services is that they must be easy for programmers to create. The main area of competition between software vendors is in the toolsets for creating WSs, and automatic code generation from WSDL description is an essential component of any such toolset.

WSDL to Java

The code we need to generate is for the SOAP client that knows how to invoke our PrimeFactors server. The program that generates such code within the Axis framework is called Wsdl2java. To simplify running it, we provide a batch file, PFStringW2J.bat, which is one long line, shown in Listing 9-3 divided into three lines and abbreviated.

Listing 9-3. Batch file to Invoke Wsdl2java

```
java -cp (classpath specification goes here)
     org.apache.axis.wsdl.Wsdl2java --verbose -o PFString
     http://localhost:8080/axis/PrimeFactorsString.jws?wsdl
```

The classpath specification simply makes sure that all the necessary libraries are available, including wsdl4j.jar and xerces.jar. The libraries are located in common\lib, a common library space for all Tomcat Web applications. Because our SOAP server and SOAP client run within the same Tomcat installation, this may be a little confusing. Conceptually, we are still on the SOAP server, running Wsdl2java to generate SOAP client code from WSDL that was generated on the server from server code. The program is asked to run in the verbose mode, place its output in the webapps/axis/PFString directory (which doesn't need to exist before this batch file is run), and get its input from the generated WSDL file.

> **NOTE** *We want to emphasize that the WSDL input can come from any WSDL, not necessarily automatically generated from a SOAP server. In a typical situation, SOAP servers and SOAP clients are created by different programmers working in different organizations. When a Web service is published (via UDDI or some other mechanism), it exposes its functionality as a WSDL description. It is this WSDL that is used to generate clients. A service programmer, in creating a WSDL for his service, may use the automatically generated description, or manually post-edit it, or write a better description from scratch.*

Looking into the PFString directory, we find that it contains a localhost subdirectory that contains three Java code files. The files define an interface and two classes within the localhost package. (The fact that the subdirectory and the Java package it contains are called “localhost” has no special significance: putting the code in a package makes a lot of sense, a package needs a name, and the host name is as good as any.) The interface and two classes are as follows:

- *interface* `PrimeFactorsStringPortType`: declares the operations that the service supports (a single `factors()` operation in our example)

- *class* `PrimeFactorsStringSoapBindingStub`: implements the PortType interface, translates a Java method call into a SOAP message using built-in Axis classes, and sends the message to the SOAP server

- *class* `PrimeFactorsString`: the SOAP client itself that gets instantiated in the JSP and supplies a server stub with a public method to call

We will look at the code in a moment; let’s use it first. To be usable, the code has to be compiled and placed where a JSP can find it, that is, common/lib. So our task is to compile the code, put it in a JAR archive, and copy the archive to common/lib. We again provide a batch file for the task, makePFString.bat, which is shown in Listing 9-4.

Listing 9-4. Batch File to Compile, Jar, and Copy the SOAP Client and Server Stub

```
cd PFString
javac -classpath (classpath specification, same as in Listing 9-2)
      localhost/*.java
jar -cf PFString.jar localhost/*.class
copy PFString.jar ..\..\..\common\lib
```

Conceptually, in terms of Web service components and their places of residence, this batch file compiles SOAP client code on the SOAP server machine and copies it to the SOAP client. As we said, because our SOAP server and SOAP client run within the same Tomcat installation, some libraries in common/lib work for the server, some others work for the client, and yet others (such as xerces.jar) work for both.

JSP for SOAP Client

The only thing left is to write a JSP that receives HTTP requests from the end user and invokes the SOAP client. Its punchline (divided in four and simplified) is this procedure call:

```
String ans;
try{
  ans=new localhost.PrimeFactorsString().           // instantiate client
    getPrimeFactorsStringPort( SOAP Server URI ).   // get server stub
    factors(12);
// invoke service
} catch( process error message }
```

We will return to this line several times later in this section. In the meantime, the entire JSP (displayed in Figure 9-3) is primeFactorsString.jsp in Listing 9-5.

Listing 9-5. JSP to Invoke the Client

```
<%@ page errorPage="../error.jsp"
%><html><head><title>PrimeFactorsString</title></head>
This JSP invokes the Axis PrimeFactorsString service.
<%
String nStr=request.getParameter("N");
if(nStr==null){
%>Sorry, a parameter "N" is required <%
} else {
  String endPoint="http://localhost:8080/axis/PrimeFactorsString.jws";
  int N=0;
  String ans=null;
  try{
    N = Integer.parseInt(nStr);
  // the punch line; Port created with a URL argument
    ans=new localhost.PrimeFactorsString().
        getPrimeFactorsStringPort(new java.net.URL(endPoint)).
        factors(N);
```

```
  %>The prime factorization of <%= N %> is <%= ans %>. <%
  }catch(Exception ex){
%>   Sorry, we cannot find the prime factors of <%= nStr %>;
  the exception is
<textarea rows="20" cols="80">
 <% ex.printStackTrace(new java.io.PrintWriter(out,true)); %>.
</textarea>
<%  }
} %>
</body></html>
```

The magical thing about this JSP is that we can replace it, and the entire SOAP client that it invokes, with a JavaScript or a C# SOAP client, and the SOAP server won't know the difference. (You will see examples later in the chapter.) To understand how the magic works, we look inside the SOAP client and server stub and watch them in operation.

The SOAP Client

Remember the JSP's punchline:

```
  try{
    ans=new localhost.PrimeFactorsString().  // instantiate client
               getPrimeFactorsStringPort(). // get server stub
               factors(12);                 // invoke service
  } catch(Exception ex){ans=ex.getMessage();}
```

The exception caught by this code is very likely to be a `java.rmi.RemoteException`. It comes from the java.rmi package, in which *RMI* stands for *Remote Method Invocation*, a Java-specific framework for distributed applications that consist of Java classes. When both the SOAP client and SOAP server are implemented in Java, it is natural to expect that SOAP messages will ride on top of RMI exchanges.

Let's first look at the PortType interface that the server stub must implement. Apart from whitespace, the code is exactly as produced by Wsdl2java, as shown in Listing 9-6.

Listing 9-6. The PortType Interface

```
/**
 * PrimeFactorsStringPortType.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */
package localhost;
public interface PrimeFactorsStringPortType extends java.rmi.Remote {
  public java.lang.String factors(int arg0) throws java.rmi.RemoteException;
}
```

The server stub must implement one method, corresponding to the only operation of the SOAP server. Its name is `factors()`, its return value is of type `String`, and it takes one argument (with the generated name `arg0`) whose data type is `int`. It is fairly easy to see how this code can be synthesized from the WSDL. The client itself is programmed exclusively in terms of this interface. This is the way that RMI operates: when an RMI component invokes a remote method, it pretends that it's a local method of the local stub class; the stub class takes care of all data interchanges between itself and the real remote class.

The client class, PrimeFactorsString, has one variable that is the URL of the service, and two methods that return an object implementing the PortType interface. One of them takes no argument and returns a server stub that connects to the server identified by the URL; the other takes a URL as an argument and returns the service identified by that URL. All these methods do is establish an HTTP connection to the SOAP server. Listing 9-7 shows the code of the SOAP client, slightly shortened and some comments replaced with our own.

Listing 9-7. The SOAP Client

```
/**
 * PrimeFactorsString.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */
package localhost;
public class PrimeFactorsString {
  // the URL of the service, obtained from WSDL
  private final java.lang.String PrimeFactorsStringPort_address =
     "http://localhost:8080/axis/PrimeFactorsString.jws";
  // material removed; two methods to return a PortType object follow:
  public localhost.PrimeFactorsStringPortType
  getPrimeFactorsStringPort() {
```

```
        java.net.URL endpoint;
         try {
           endpoint = new java.net.URL(PrimeFactorsStringPort_address);
         }
         catch (java.net.MalformedURLException e) {
             return null; // unlikely as URL was validated in Wsdl2java
         }
         return getPrimeFactorsStringPort(endpoint);
  }
  public localhost.PrimeFactorsStringPortType
  getPrimeFactorsStringPort(java.net.URL portAddress) {
         try {
           return new localhost.PrimeFactorsStringSoapBindingStub(portAddress);
         }
         catch (org.apache.axis.AxisFault e) {
             return null; // ???
         }
  }
}
```

Apart from `AxisFault`, there is nothing specific to Axis or indeed specific to Web services in this code. All this is in the server stub that implements the PortType interface. `AxisFault` is a class that extends `java.rmi.RemoteException` with additional fields and methods to provide access to SOAP-level error conditions called *faults.* We will discuss SOAP faults in the SOAP overview section.

The Server Stub

The server stub makes extensive use of the Axis framework. We are not going to present all 95 lines of its code, but will try to show enough to indicate the main components and actions.

The code (PrimeFactorsStringSoapBindingStub.java) starts out by declaring two private variables that give a clue to where the main functionality of the class is hidden. (See Listing 9-8.)

Listing 9-8. The Stub's Variables: Service and Call

```
public class PrimeFactorsStringSoapBindingStub
    extends javax.xml.rpc.Stub
    implements localhost.PrimeFactorsStringPortType {
  private org.apache.axis.client.Service service = null ;
  private org.apache.axis.client.Call call = null ;
```

Evidently, there is an `org.apache.axis.client` package that contains `Service` and `Call` classes. The default constructor of our class shows how Service and Call are related. (See Listing 9-9.)

Listing 9-9. The Stub's Default Constructor

```
public PrimeFactorsStringSoapBindingStub()
    throws org.apache.axis.AxisFault {
  try {
    service = new org.apache.axis.client.Service();
    call = (org.apache.axis.client.Call) service.createCall();
  }
  catch(Exception t) {
    throw org.apache.axis.AxisFault.makeFault(t);
  }
}
```

However, tracing the path of calls that, in the end, creates the stub, we discover that it is not created by the default constructor, but rather by a constructor with a `java.net.URL` argument. The path starts in the JSP, where the punchline (again) says

```
ans=new localhost.PrimeFactorsString().
    getPrimeFactorsStringPort(new java.net.URL(endPoint)).
    factors(N);
```

The `getPrimeFactorsStringPort()` method within the SOAP client, when called with a URL argument, calls the constructor with that argument:

```
return new localhost.PrimeFactorsStringSoapBindingStub(portAddress);
```

Back in the stub class, we can see that it does indeed have another constructor, as shown in Listing 9-10.

Listing 9-10. The Stub's `java.net.URL` *Constructor*

```
public PrimeFactorsStringSoapBindingStub(java.net.URL endpointURL)
      throws org.apache.axis.AxisFault {
    this(); // call the default constructor, creating Service and Call
    // now set the properties of the Call object
    call.setTargetEndpointAddress( endpointURL );
    call.setProperty(org.apache.axis.transport.http.HTTPTransport.URL,
                     endpointURL.toString());
}
```

Ignoring the details, we can say that the Call object emerges from the constructor completely aware of the location of the service it is going to call, which is not really surprising. Ignoring more details, we can look at the `factors()` method (Listing 9-11).

Listing 9-11. The Stub's `factors()` *Method*

```
public java.lang.String factors(int arg0) throws java.rmi.RemoteException{
    if (call.getProperty(
           org.apache.axis.transport.http.HTTPTransport.URL
                          )==null) {
      throw new org.apache.axis.NoEndPointException();
    }
    call.removeAllParameters();
  // set the call's parameter within the SOAP message, xsd:int
    call.addParameter("arg0", new org.apache.axis.encoding.XMLType(
              new javax.xml.rpc.namespace.QName
                ("http://www.w3.org/2001/XMLSchema","int")),
      org.apache.axis.client.Call.PARAM_MODE_IN);
  // set the call's return type within the SOAP message, xsd:string
    call.setReturnType(new org.apache.axis.encoding.XMLType(
      new javax.xml.rpc.namespace.QName("http://www.w3.org/2001/XMLSchema",
                                          "string")));
  // ACTION and NAMESPACE properties are set to ""; see SOAP section later
    call.setProperty(org.apache.axis.transport.http.HTTPTransport.ACTION, "");
    call.setProperty(call.NAMESPACE, "");
    call.setOperationName( "factors");
  // all set; invoke the service
  // the arguments are an array of Object; the returned value is Object

    Object resp = call.invoke(new Object[] {new Integer(arg0)});

    if (resp instanceof java.rmi.RemoteException) {
      throw (java.rmi.RemoteException)resp;
    }
    else { // convert returned Object to String and return
       return (java.lang.String) resp;
    }
  }
```

The punchline of this method is unquestionably the highlighted `invoke()` call. There are some Java specifics in it: because it ought to be prepared to handle objects of any class, it takes an Object array as argument and returns an object; upon receipt, the returned object is cast down to a specific class. Otherwise, it is

clearly inside this call that SOAP messages are sent and received. We could dig more deeply into the Java code of `invoke()` (it's open source!) to see what's going on, but this would take us deeper into the specific Axis framework and away from the general platform- and framework-independent operation of Web services. So, instead of reading more Java code (which we urge you to do), we are going to snoop on the SOAP messages that `invoke()` sends and receives.

TCPMon and the Messages

Axis provides a utility, TCPMon (TCP monitor), that allows you to watch HTTP traffic to a specific server and port. Suppose that we want to watch the traffic on `localhost:8080` (as is indeed the case). Instead of specifying `localhost:8080` as the target port for our traffic, we direct it to a TCPMon listener on another host and/or port (say, `localhost:8081`), and instruct the listener to pass on the 8081 traffic to *its* target host and port, which we set to `localhost:8080`. Now all messages to and from `localhost:8080` can be intercepted and displayed by the TCPMon listener on `localhost:8081`. This is usually called "HTTP tunneling": TCPMon serves as a tunnel (with a hidden video camera) that lets HTTP traffic pass through.

> **NOTE** *There is nothing SOAP-specific about TCPMon and HTTP tunneling in general. You can establish a TCPMon mapping from any local port to any web server, either local or nonlocal, serving either static or generated pages. For instance, you can associate port 8082 with* `www.google.com:80`*, connect to* `localhost:8082`*, use the engine, and have TCPMon display your queries going out and HTML coming back.*

The details of TCPMon operation can be found in the Axis User Guide. We provide a batch file, webapps/axis/runTCPMon.bat, that opens the TCPMon Admin panel. In that panel, enter "8081" in the Listen Port# box, enter "localhost" under Listener Target Hostname, and "8080" under Listener Target Port#. Click on the Add button. To watch the traffic, click on the newly added Port 8081 panel.

Assuming that we have set up a listener as described in the preceding paragraph, all that's left to do is go into primeFactorsString.jsp in xmlp/ws and change the value of its `endPoint` variable:

```
String endPoint="http://localhost:8081/axis/PrimeFactorsString.jws";
```

As we explained in the preceding section, this endpoint will end up, as a URL object, in the server stub that will use it to invoke the service. Because our SOAP messages ride on top of HTTP, invoking a service means sending an HTTP

request (that contains a SOAP request) and receiving back an HTTP response (that contains a SOAP response). TCPMon will obligingly display all these messages for us. Let's look at the SOAP request first. (See Listing 9-12.) It happens to be an HTTP POST request whose body is a SOAP message and the headers include one nonstandard `SOAPAction` header.

Listing 9-12. SOAP Request Within HTTP POST Request

```
POST /axis/PrimeFactorsString.jws HTTP/1.0
Content-Length: 402
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <factors>
   <arg0 xsi:type="xsd:int">36</arg0>
  </factors>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We observe that the root of a SOAP message (viewed as an XML document) is an `Envelope` element that has a single child, a `Body` element. Both are in the "soap envelope" namespace. The other two declared namespaces are for XSchema and XSchema Instance. They are used to declare data types, as in `xsi:type=xsd:int`. (Recall from the preceding chapter that the `xsi:type` attribute is for declaring types outside any schema context.) The contents of the `Body` element are the remote procedure call. They could be in a namespace of their own, but, in this message, they are not in any namespace. Recall that the WSDL of Listing 9-2 had `<soap:body...namespace="".../>`, and correspondingly, in Listing 9-11, `call.NAMESPACE` was set to "".

The convention for passing a procedure call in a SOAP message is that the name of the procedure is the tagname of a child element of the body, and the arguments are children of that element, named `arg0`, `arg1`, and so on if they are automatically generated. (In a handcrafted SOAP message, you can give them any names you want.) The content of an argument element is its value, and each of them can have an `xsi:type` element to indicate its data type.

> **NOTE** *Another, more structured method of attaching types to elements would be to place the contents of the body in a namespace and associate that namespace with an XSchema (or, indeed, a RELAX NG grammar) that would define the data types of elements.*

Let us look at the response (Listing 9-13), to see if it conforms to the same pattern.

Listing 9-13. SOAP Response Within HTTP POST Request

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 444
Date: Sun, 10 Feb 2002 22:19:37 GMT
Server: Apache Tomcat/4.0.1 (HTTP/1.1 Connector)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  (same namespaces and the encoding attribute)
>
 <SOAP-ENV:Body>
  <factorsResponse>
   <factorsResult xsi:type="xsd:string">2*2*3*3</factorsResult>
  </factorsResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Indeed, we see the same message structure and the same use of XS2 to indicate data types. The names of elements within `Body` follow obvious naming conventions.

The SOAP messages of Listing 9-12 and 9-13 completely define the operation of the service. Any SOAP client that sends the HTTP request of Listing 9-12 will receive the response of Listing 9-13. (If our SOAP server were set up to work over SMTP instead of HTTP, any SOAP client that's capable of sending an email with the HTTP body of Listing 9-12 would receive back an email with the HTTP body of Listing 9-13.) We are going to illustrate this point by writing two alternative clients in different languages that will communicate with the same server using the same SOAP messages.

Client Variations

In this section, we are not going to use any frameworks or toolsets but rather manufacture SOAP messages directly from HTML form data. The idea is that a SOAP request is an XML document, which, in turn, is a character string, which, in turn, is a sequence of bytes. The SOAP client doesn't need to know anything about SOAP: it can construct the SOAP request as a character string and deliver it to the SOAP server as an XML document or as a sequence of bytes: the operation of the SOAP server will remain unaffected.

Variation 1: SOAP as XML

Our first alternative client will use Microsoft-specific JScript code within Internet Explorer. It will construct a SOAP request as text, parse it into an MSXML DOMDocument object, and send it directly to the SOAP server using an XML-HTTP ActiveX object. It will receive a SOAP response back as an XML document and display it with the help of an XSLT. This is illustrated in Figure 9-5.

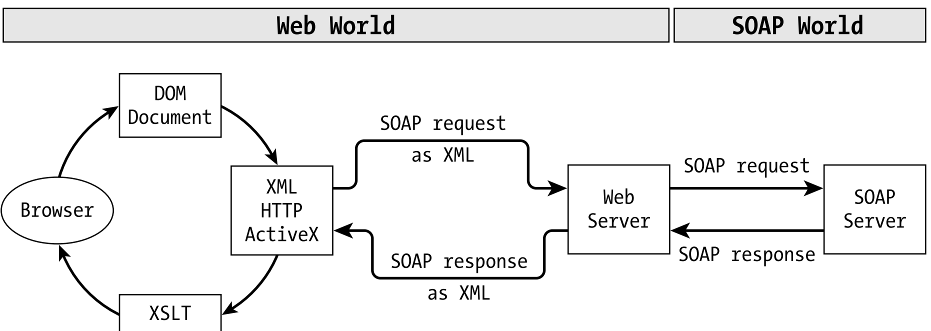

Figure 9-5. Client variation 1: SOAP as XML

The HTML Page

Our first no-framework example is an HTML page with some JScript code and a form, primeFactorsStringDirect.htm. The code initializes four variables (see Listing 9-15) and defines three functions:

```
function factorSoapEnv(N){ // build the SOAP request envelope
function getFactors(N){ // invoke the service
function onSub(){ // onSubmit action: call getFactors()
```

The body of the page consists entirely of the form. The form has an input element to enter a number to factor, and a `<div>` to receive the result. The result is computed by the JScript `getFactors()` function that talks directly to the SOAP server, as shown in Listing 9-14.

Listing 9-14. Another Kind of Client

```
<html><head><title>primeFactorsStringDirect.htm</title>
<script LANGUAGE="JScript">...</script></head>
<body><form name="theForm"
  action="javascript:void"
  onSubmit="return onSub()">
 <input type="button" value="getFactors"
  onclick="resultDiv.innerHTML=getFactors(theForm.N.value)"/>
 <br/>number to factor:
  <input type="text" name="N" size="20" value="120">
 <br/>result: <div ID="resultDiv"> </div>
</form> </body></html>
```

Submitting the form results in the same action as the button's `on`:

```
function onSub(){
  resultDiv.innerHTML=getFactors(theForm.N.value);
  return false;
}
```

The JScript Code

Obviously, all the action is in the `getFactors()` function. How does a JavaScript function invoke a SOAP action? It uses four ActiveX objects in the background, as shown in Listing 9-15.

Listing 9-15. Four ActiveX Objects

```
  var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
  var SOAPRequest = new ActiveXObject("MSXML2.DOMDocument");
  SOAPRequest.async = false;
  var SOAPResponse = new ActiveXObject("MSXML2.DOMDocument");
  SOAPResponse.async = false;
  var objStyle = new ActiveXObject("MSXML2.DOMDocument");
  objStyle.load("primeFactorsStringDirect.xsl");
  objStyle.async = false;
```

The first of these objects, `XMLHTTP`, has not been used in this book before. It is aware of XML and HTTP, so it has methods for setting headers, loading a body that is an XML document, sending out an HTTP request, and receiving an HTTP response. The `SOAPRequest` and `SOAPResponse` objects are familiar DOMDocuments, which will be used as holders for SOAP messages which, for those DOMDocument objects will be just XML data. Finally, `objStyle` is a DOM object that holds an XSLT for transforming a SOAP message into HTML. Perhaps the most educational feature of this example is that the same DOMDocument object is used to process SOAP messages and XSLTs because they are all XML data.

We also need a supporting function that will take the value of the `N` parameter and construct a SOAP envelope. This is a pedestrian kind of function that keeps on concatenating the right strings. A number of service-specific features are hard-wired into its code, as shown in Listing 9-16.

Listing 9-16. Function to Build an Envelope (Very ad hoc)

```
function factorSoapEnv(N){ // build the SOAP request envelope
 var S='<?xml version="1.0" encoding="UTF-8"?>\n';
 S+='<SOAP-ENV:Envelope\n';
    S+='SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"\n';
    S+='xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"\n';
    S+='xmlns:xsd="http://www.w3.org/2001/XMLSchema"\n';
    S+='xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">\n';
    S+='<SOAP-ENV:Body>\n';
    S+='<factors>\n';
    S+=' <arg0 xsi:type="xsd:int">'+N+'</arg0>\n';
    S+='</factors>\n';
   S+='</SOAP-ENV:Body>\n';
 S+='</SOAP-ENV:Envelope>\n';
 return S;
}
```

With the supporting function and ActiveX objects in hand, we can write `getFactors()`, shown in Listing 9-17. It is completely obvious except for one annoying detail: to get it to work, we have to specify (redundantly) the value of the `SOAPAction` header. (In the Axis version of the client, it was set to the empty string.)

```
xmlhttp.setRequestHeader("SOAPAction", "PrimeFactorsString")
```

This is something that is required by Microsoft software and is likely to change in the future because the header goes back to SOAP 1.1 and is deprecated in SOAP 1.2.

Listing 9-17. The `getFactors()` *Function*

```
function getFactors(N){
  // open connection to the SOAP server URL
  xmlhttp.Open("POST",
               "http://localhost:8080/axis/PrimeFactorsString.jws",
               false); // false stands for "asynchronous=false"
  // set HTTP headers including SOAPAction
  xmlhttp.setRequestHeader("SOAPAction", "PrimeFactorsString")
  xmlhttp.setRequestHeader("Content-Type", "text/xml; charset=utf-8")
  // load SOAP request constructed by factorSoapEnv()
  SOAPRequest.loadXML(factorSoapEnv(N));
  // send the HTTP request out
  xmlhttp.Send(SOAPRequest.xml);
  // load the result (serialized XML) into another DOM document
  SOAPResponse.loadXML(xmlhttp.responseXML.xml);
  // send the SOAP response, as DOM, to a trasformer object
  // to convert to HTML
  var result=SOAPResponse.transformNode(objStyle.documentElement);
  return result;
}
```

The last thing to consider is the `objStyle` stylesheet for transforming a SOAP response message into HTML. This will require an analysis of possible returned values and a discussion of SOAP faults.

SOAP Returned Values and an XSLT to Display Them

If our SOAP communication is successful, we'll get back the familiar SOAP response of Listing 9-13. Its body is

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  (namespaces and the encodingStyle attribute)
>
 <SOAP-ENV:Body>
  <factorsResponse>
   <factorsResult xsi:type="xsd:string">2*2*2*3*5</factorsResult>
  </factorsResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

From this, our XSLT will extract the data type of the result and its value. However, if SOAP communication fails, the `Body` element will contain something else: one or more `Fault` elements, also in the `SOAP-ENV` namespace. In this case, our stylesheet will extract the entire Body content and place it into a text area. Because in either case we are interested in the content of the `Body` element, our root template will use `apply-templates` to select `Body`'s children, and the other two templates will process the two possible versions of its content. (See Listing 9-18.)

Listing 9-18. XSLT to Display Result

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   exclude-result-prefixes="xsi xsd SOAP-ENV"
>
<xsl:output method="html"/>
<xsl:template match="/">
  <xsl:apply-templates select="SOAP-ENV:Envelope/SOAP-ENV:Body/*" />
</xsl:template>
<xsl:template match="factorsResult"><!-- output success -->
  <p>
  <em>type</em>=<xsl:value-of select="@xsi:type"/>;
  <em>value</em>=<xsl:value-of select="."/><br/>
  </p>
</xsl:template>
<xsl:template match="SOAP-ENV:Fault"><!-- output failure -->
  <p><H1>ERROR</H1>
  <em>code</em>=<xsl:value-of select="SOAP-ENV:faultcode"/>; <br/>
  <em>short version</em>=<xsl:value-of select="SOAP-ENV:faultstring"/>; <br/>
  <em>details:</em><br/>
  <textarea rows="40" cols="60">
    <xsl:copy-of select="SOAP-ENV:detail"/>
  </textarea>
  </p>
</xsl:template>
```

We will discuss the details of SOAP fault elements in a moment, after we present our second SOAP client variation.

Variation 2: SOAP As Bytes

Our second alternative client uses standard HTML and ECMAScript to construct a SOAP request and send it to a JSP, along with the SOAP server URL. The JSP (which does not use any Axis-specific code and so can easily be replaced by an ASP) opens a direct socket connection to the SOAP server and passes along the SOAP message as a stream of bytes. It receives the SOAP response also as a stream of bytes, and passes it on back to the browser. This is summarized in Figure 9-6.

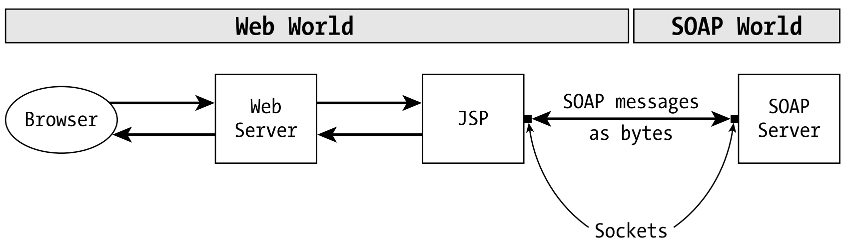

Figure 9-6. Client variation 2: SOAP as a stream of bytes

The HTML Page

The HTML page of this client, primeFactorsString.htm, is very similar to Listing 9-14 but simpler. (See Listing 9-19.) It constructs the SOAP envelope exactly as in Listing 9-14 but sends it to a JSP page for further processing rather than directly to the SOAP server. The form in the page does have two additional hidden input elements: one to submit the URL of the SOAP server and the other to submit the SOAP envelope constructed by a JavaScript function.

Listing 9-19. The HTML Page That Submits Data to PassAlong.jsp

```
<html><head><title>primeFactorsString.htm</title></head><body>
<script>...</script>
<form name="theForm" method="POST" action="PassAlong.jsp"
    onSubmit="return onFactorSub()">
number to factor:<input type="text" name="N" size="20" value="120" >
<input type="hidden" name="router"
    value="http://localhost:8080/axis/PrimeFactorsString.jws" >
<input type="hidden" name="envelope" value="" >
<input type="submit">
</form></body></html>
```

The form's `onSubmit` action is simply to store the SOAP envelope (constructed by JavaScript) in the envelope hidden field and submit the form:

```
function onFactorSub(){
  theForm.envelope.value=factorSoapEnv(theForm.N.value);
  return true;
}
```

The `factorSoapEnv()` function is identical to the one of the same name in the preceding version of the client (Listing 9-16): it just keeps on concatenating all the right strings until the envelope is ready to go. At this point, the action shifts to PassAlong.jsp.

The JSP and the Socket Connection

The JSP receives two items from the browser: the URL of a SOAP server and a SOAP envelope to send. In outline, it proceeds as follows:

1. Take the URL apart and open a socket connection to the server, and set up streams to send and receive data.
2. Construct headers for the SOAP request.
3. Send the headers and the SOAP envelope to the SOAP server.
4. Receive the response as a stream of (integer codes for) characters, and convert to a text string (the result string). Close the SOAP server connection.
5. Send the result string, without its HTTP headers, back to the browser.

We will present PassAlong.jsp in two installments, Listing 9-20 and 9-21.

Listing 9-20. Receive HTTP Request; Send SOAP Request

```
<%@ page  errorPage="error.jsp" import="java.net.*,java.io.*"
 %><%
  String urlString=request.getParameter("router");
  String payload=request.getParameter("envelope");
  URL url=new URL(urlString);
  int timeout=20000; // 20 seconds
  int port= url.getPort(); if(port<0) port=80;
// open socket connection to SOAP server; set up streams
```

```
  Socket s = new Socket(url.getHost(),port);
  s.setSoTimeout(timeout);
  OutputStream outStream = s.getOutputStream ();
  InputStream inStream = s.getInputStream ();
// construct HTTP headers for SOAP request
  StringBuffer headerbuf = new StringBuffer();
  headerbuf.append("POST ")
    .append(url.getFile()).append(" HTTP/1.0\r\n")
    .append("Host: ")
       .append(url.getHost()).append(':').append(port).append("\r\n")
    .append("Content-Type: text/xml; charset=utf-8\r\n")
    .append("Content-Length: ")
      .append(payload.length()).append("\r\n")
    .append("SOAPAction: \"\"\r\n") // the empty string
    .append("\r\n"); // end of HTTP headers; a blank line
// Send HTTP headers and payload downstream to SOAP server
  BufferedOutputStream bOutStream = new BufferedOutputStream(outStream);
  bOutStream.write(headerbuf.toString().getBytes("utf-8"));
  bOutStream.write(payload.getBytes("utf-8"));
  bOutStream.flush(); outStream.flush();
```

The next listing (Listing 9-21) shows the processing of the SOAP response. The response comes back over the byte stream `inStream` attached to the remote socket. We want to convert this into a character string, to be sent back to the browser. This involves several transformations. First, we buffer the byte stream. Second, we wrap the byte stream, which is, in fact, a UTF-8 encoding of Unicode characters, into a character stream, specifying the encoding. However, Java character streams do not really carry values of type `char`; they carry integers that are Unicode codes for the character data, appropriately encoded (UTF-8 in this JSP). So we have to convert those Unicode codes into real characters before storing them in a Java String object. This involves copying them to a StringWriter stream, converting from `int` to `char` in the process. When this is done, the StringWriter content is easily dumped to a string.

That string contains both the SOAP response message and the HTTP headers for carrying it over HTTP. We chop off the headers and send the response message back to the browser. We could easily have piped it through an XSLT, as in the preceding example.

Listing 9-21. Receive SOAP Request; Process; Send to Browser

```
// Prepare InputStreamReader
  BufferedInputStream bInStream = new BufferedInputStream(inStream);
  InputStreamReader reader=new InputStreamReader(bInStream,"utf-8");
// convert incoming stream of "characters as integers" into String
  StringWriter sw=new java.io.StringWriter();
  for(int ch=reader.read();ch>=0;ch=reader.read())
    sw.write((char)ch); // convert int to char, write to StringWriter
  String resString=sw.toString();
// close up SOAP server connection
  bOutStream.close(); outStream.close();
  bInStream.close();  inStream.close();
  s.close();
// chop off headers from the result string
  int endHeaderPos=resString.indexOf("\r\n\r\n");
  if(endHeaderPos>=0) resString=resString.substring(endHeaderPos+4);
// send result string back to the browser
  response.setContentType("text/xml; charset=utf-8");
  response.setContentLength(resString.length());
  out.write(resString);
 %>
```

This completes PassAlong.jsp, the Client Variations section, and the entire PrimeFactorsString example. The code archive also contains the PrimeFactorsArray example that you saw displayed in Figure 9-3. It is very similar to PrimeFactorsString, except the Web service returns prime factors as an array of integers. The SOAP server code is in axis/PrimeFactorsArray.jws, and the SOAP client is invoked by xmlp/ws/primeFactorsArray.jsp. The only novelty is that the generated code returns a value of type `Object[]`, and our JSP that invokes the client and receives the result has to do some type conversions. Sometimes when you're going to depend on generated code, you do have to read it.

Overview of SOAP 1.2

Now that we have seen several examples of SOAP messages, we can take a general look at the specification that defines them. As mentioned, it consists of three parts: the main Part 1, Part 0 (Primer), and Part 2 (Adjuncts). Before version 1.2, Part 0 didn't exist and Part 2 was included in the core specification.

Part 1 consists of the following sections (listed in order of size and significance):

- The XML structure of the SOAP envelope, by far the longest and most important
- SOAP Message Exchange Model, mostly relevant for somewhat distant future
- SOAP Protocol Binding Framework, general principles only (a specific proposal is in Part 2)
- Two very brief supporting sections on SOAP's relationship to XML (it *is* an XML language) and the role of URIs in SOAP (important)

With much of earlier SOAP relegated to Adjuncts, Part 1 feels a little too general and abstract.

Part 2 has three major sections:

- SOAP encoding, that is, a set of specific rules for encoding data types in XML
- Using SOAP for RPC, that is, a set of conventions for encoding a procedure call in a SOAP envelope structure
- Default SOAP HTTP binding

As we mentioned, the first two of these sections have been made into Adjuncts to give more flexibility to individual developers and applications. The status of the third section is as follows: you can use any protocol, but most likely you will use HTTP; if you use HTTP, you can embed SOAP envelopes into HTTP exchanges in any way you wish, but here is a default binding that will most likely be understood without any additional arrangements. So far, most Web services have been using the default HTTP, but there is active experimentation with SMTP and instant messaging as alternatives.

In this section, we will discuss the Message Exchange Model and the XML structure of the SOAP envelope. In the next section, we will discuss SOAP encoding and SOAP RPC conventions, in the context of an RPC example that returns structured data (an array of objects) rather than a simple type, as in earlier examples.

In the remainder of this chapter, we will say *SOAP1.2-1* to mean *SOAP 1.2 Working Draft Part 1*, and similarly for *SOAP1.2-2* and *SOAP1.2-0*.

SOAP Message Exchange Model

A SOAP message is a one-way transmission from a SOAP sender to a SOAP receiver. However, SOAP messages can be combined to implement various Message Exchange Patterns (MEPs) such as request/response or multicast. The definition of a MEP would describe

- the life cycle of an exchange conforming to the pattern within a specific transport protocol,
- the temporal and causal relationships of the messages within the pattern, and
- the terminating conditions of the pattern, both normal and abnormal.

SOAP1.2-2 gives a general definition of MEPs and a specific definition of just one of them, the "Single-Request-Response" MEP.

SOAP messages can travel from the message originator to its final destination via intermediaries that can simply pass the message on or process it and emit a modified message or a fault condition. *SOAP node* is a general name for the initial SOAP sender, the ultimate SOAP receiver, or a SOAP intermediary (which is both a SOAP sender and a SOAP receiver). Ultimately, a Web service is a collection of SOAP nodes.

A SOAP message consists of two parts: the optional header and the mandatory body. Both header and body consist of "blocks," which are XML elements. The content of the body, sometimes called the *payload*, is intended to be processed by the message's final destination. The content of the header is intended for intermediate nodes, with each header block targeted individually. The entire machinery of header blocks and intermediate nodes is intended for automatic collaboration among multiple SOAP processors. For instance, a purchase order from a SOAP client to a SOAP server may travel via an intermediate node (specified in a header block) that will authenticate the digital signature of the originator of the purchase order, and another intermediate node that will initiate just-in-time delivery of the ordered items. As we said, this functionality is probably some time away in the future.

The XML Structure of a SOAP Message

This section explains how the abstract structure of a SOAP message is expressed in XML.

The Root Element and Its Schema Definition

SOAP1.2-1 and *SOAP1.2-2* use the infoset terminology to describe XML data. So, for instance, a SOAP message is defined in *SOAP1.2-1* Section 4 as follows: "A SOAP message has an XML infoset that consists of a document information item with exactly one child, which is an element information item as described below." Continuing the quote, that "element information item" has the following properties:

- a local name of `Envelope`
- a namespace name of `http://www.w3.org/2001/12/soap-envelope`
- zero or more namespace-qualified attribute information items
- one or two element information item children in order as follows:
  - an optional `Header` element information item (see 4.2 SOAP Header)
  - a mandatory `Body` element information item (see 4.3 SOAP Body)

We are going to use looser (but less verbose) terminology of *document*, *element*, and *attribute*. So we recast the official definition as follows.

A SOAP message is an XML document. Its root element's name is `Envelope`, within the `http://www.w3.org/2001/12/soap-envelope` namespace. (In our discussion, we will call it the *Envelope namespace* and assume that it is mapped to the `env` prefix.) The `env:Envelope` element has two children in the same namespace: the optional `env:Header` and the required `env:Body`. It may also have other children as long as they come from other namespaces, and it may have global (that is, namespace-qualified) attributes.

To make it even clearer, here is an excerpt from the XSchema for `Envelope`, `Header`, and `Body` elements (`http://schemas.xmlsoap.org/soap/envelope/`). This is SOAP 1.1, but the changes for 1.2 are minimal and listed in *SOAP1.2-1*. The schema is very readable, with only two details possibly unfamiliar: `namespace=##other` means "any namespace other than currently in scope," and `processContents="lax"` means "don't give me a hard time with this one." (See Listing 9-22.)

Listing 9-22. XSchema for SOAP Envelope

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/" >
  <xs:element name="Envelope" type="tns:Envelope" />
  <xs:complexType name="Envelope" >
    <xs:sequence>
      <xs:element ref="tns:Header" minOccurs="0" />
      <xs:element ref="tns:Body" minOccurs="1" />
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
                                        processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>
```

Next, we look at the `Header` and `Body` elements and other items defined in the Envelope namespace.

The Header Element

Recall that, in the abstract Message Exchange Model, both header and body consist of blocks. In XML terms, both `env:Header` and `env:Body` can have any number of children elements in their own namespace(s), with no constraints on their internal structure. Header blocks, as we said, can be independently targeted at intermediate SOAP nodes. To express such targeting in XML, *SOAP1.2-1* defines three attributes in the Envelope namespace: `actor`, `mustUnderstand`, and `encodingStyle`.

The value of `actor` can be any URI. SOAP nodes have a property called `role`, whose value is also a URI; if the value of `actor` on a header block matches the value of `role`, then the block is targeted at that node. *SOAP1.2-1* does not explain how a node specifies its role, but it does mention, in Section 2.2, that "each SOAP node MUST act in the role of the special SOAP actor named `http://www.w3.org/2001/12/soap-envelope/actor/next`."

If the value of `actor` is not specified, the block is targeted at the ultimate receiver of the message.

If the node's role matches a header block's `actor` attribute and, in addition, the value of `mustUnderstand` on that header block is `1` or `true`, then the node must either process the block or emit an error message.

The `encodingStyle` attribute, which you have seen in our examples, specifies the XML encoding of data items. Its value is a URI. For the encoding defined in *SOAP1.2-2*, that URI is `http://schemas.xmlsoap.org/soap/encoding/`, but it can be any URI that serves as an identifier for an application-specific encoding.

Listing 9-23 provides an example from *SOAP1.2-0* that illustrates this usage, as well as the `actor` and `mustUnderstand` attributes.

Listing 9-23. A Header Example

```
<env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
         env:actor="http://www.w3.org/2001/12/soap-envelope/actor/next"
         env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</reference>
   <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>
  </m:reservation>
  <t:transaction
          xmlns:t="http://thirdparty.example.org/transaction"
          env:encodingStyle="http://example.com/encoding"
          env:mustUnderstand="true" >
              5
  </t:transaction>
</env:Header>
```

This example has two header blocks. The first header block must be processed by the first SOAP node it encounters, and the second header block is targeted at the ultimate receiver. The content of the node is a piece of data to be processed by the targeted node in accordance with the encoding identified by `http://example.com/encoding`.

We have not yet seen or read about a working example that actually uses header blocks in this way.

The `Body` *and* `Fault` *Elements*

The `env:Body` element never carries an `env:actor` attribute because it is targeted at the ultimate receiver of the message. It can have any number of children, of two different kinds. Under normal conditions, children of `env:Body` must be elements from namespaces other than the Envelope namespace. (The requirement that children of `env:Body` must be in *some* namespace is new in 1.2 and still under discussion.) If a SOAP error occurs, `env:Body` may have any number of `env:Fault` elements. (Recall that we wrote code for this possibility in the XSLT of Listing 9-18.)

The `env:Fault` element has two mandatory children and two optional ones. The mandatory children are `faultcode` and `faultstring`, both in no namespace.

The optional children are `faultfactor` and `detail`, also in no namespace. Their use is as follows:

- `faultcode` is for use by software. Their value is a qualified name in the Envelope namespace. *SOAP1.2-1* Section 4.4.5 defines half a dozen `faultcode` values, such as `env:VersionMismatch` and `env:DataEncodingUnknown`.
- `faultstring` is to provide a human-readable description of the fault.
- `faultfactor` is a URI, typically one of the URIs that is used as a value for an `actor` attribute. It serves to provide information about which SOAP node on the SOAP message path caused the fault. In the multinode situation, intermediate nodes generating an error MUST emit a `faultfactor` element; the ultimate receiver may, but doesn't have to do so, because . . .
- `detail` is intended for error information related to `env:Body`. If the contents of the SOAP body could not be processed successfully, `detail` *must* be present within `env:Fault`.

This concludes our discussion of *SOAP1.2-1*. By the time you read this, some details of our discussion may be out of date, but we have tried to concentrate on those aspects of the specification that seem most stable and useful in practical programming. We move on to *SOAP1.2-2*, and specifically to encoding and RPC conventions.

XML Encoding and RPC Conventions

In this section, we will be talking specifically about using SOAP and Web services for RPC, but the principles of XML encoding apply to other scenarios as well.

All our previous examples have shown RPCs of the simplest possible kind, in which both the argument of the procedure call and the returned value are of a primitive type. In other words, both the SOAP request and the SOAP response have to encode only a single primitive value, which can be done by an `xsi:type` attribute.

In the general case, both the argument and the returned values are complex structures with subparts and references to them. (Note that one argument is enough: if there are more, we can always wrap them in a structure of which they are subparts.) So, for the general case, we need a set of general conventions for serializing a complex structure with references to its subparts as an XML structure within a SOAP message.

An Example

We will illustrate the notion of structure with subparts on a simple example: an inventory of office equipment that lists such items as computers, printers, scanners, and fax machines. It is intended purely as an illustration of concepts. Once the concepts are in place, we will develop another example that we will implement as a real Web service.

Each item in our inventory is represented by a structure with three fields: a unique tag number, a character string that classifies the item, and the item's price. In C, we would define such as structure as follows:

```
typedef struct{
  int tagNumber;
  char * classifier;
  double price;
}Item;
```

The reason we are using C rather than Java for the illustration is because the C `struct` is much closer in some respects to XML representations than are Java/C++ objects. The C `struct` usually has only data, not methods (although it can, of course, contain pointers to functions), and it has no notion of controlled access: everything is public. In fact, the XML encoding of *SOAP1.2-2* uses the `struct` element to represent structured objects whose subparts are referenced by name. (It uses the `array` element to represent structured objects whose subparts are addressed by a numerical index.)

With the structure defined, we can describe individual items (still using C syntax):

```
Item i1 = { 1, "computer", 679.95 }
Item i3 = { 3, "printer", 122.50 }
```

An item can combine more than one function; for instance:

```
Item i6 = { 6, "printer scanner fax", 400.00 }
```

An office inventory is itself a structure that has four subparts: `computer`, `printer`, `scanner`, and `faxMachine`. An office is allowed at most one of each. A fully equipped office will look like

```
Office o1 = { i1, i3, i4, i5 } // i4 is scanner, i5 is fax machine
```

An office that has a multifunction device will look like

```
Office o2 = { i2, i6, i6, i6 } // i2 is computer
```

Notice that a single `struct`, `i6`, is referenced more than once. If we think of each reference as a directed link, the entire structure forms a directed graph, shown in Figure 9-7.

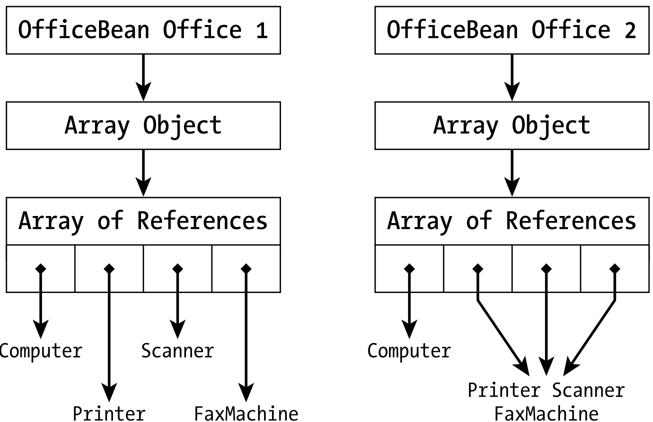

Figure 9-7. Directed graph of office items

Suppose that we want to write a Web service RPC that receives an office inventory as its argument and returns the total price of all equipment in that office. The task of XML encoding would be to represent an arbitrary directed graph of structures and references in XML. In particular, the XML representation would have to represent the fact that, in our second office, the price of the multifunction device must be counted only once. The way this is done is what is meant by the "SOAP encoding."

SOAP Encoding and the Data Model

The initial SOAP authors could have, of course, used XLink for SOAP encoding, but they didn't. Instead, they designed it as a specialized XML language that is more in tune with the notions that are traditionally used in programming languages: `struct` and `array`. This language has been carried over from earlier versions into Section 4 of *SOAP1.2-2*, with three changes (two of which are promised but not yet implemented).

The first change is a change in status: the use of the encoding style as defined in *SOAP1.2-2* is "encouraged but not mandatory." Individual applications can use their own serialization rules.

The second change is conceptual: serialization rules are going to be separated from the data model. The "SOAP Encoding" section of *SOAP1.2-2* is preceded by a "SOAP Data Model" section, whose entire content is the following editorial comment:

> *Section 4 currently defines a data model in the form of a directed graph. Elements of the data model include struct, array, and id/href. In addition to the data model, section 4 includes a particular encoding of that model without clearly separating the two. The W[orking]G[roup] would like to clarify the relationship between the data model and the particular encoding by saying that the SOAP encoding is one of several potential encodings of the SOAP data model. This section is the placeholder for the description of the SOAP data model.*

The third change is terminological and also described in an editorial note (in the very beginning of Section 4): "The Working Group is aware that the following section does not use the XML Infoset terminology used elsewhere in this specification, and most notably in Part 1. The WG expects to rewrite this section using the XML Infoset terminology at a later date."

With these caveats and promised changes out of the way, we can quickly review the encoding rules of *SOAP1.2-2*, Section 4.

Values, Types, and Encoding

XML encoding deals with very familiar programming concepts. First, we distinguish values and types. Within values, we distinguish simple values (without subparts) and compound values (with subparts). There are two kinds of compound values: structs, whose subparts are referenced and accessed by name, and arrays, whose subparts are referenced by a numerical index from a continuous range starting at 0. Within types, we also distinguish simple types (classes of simple values) and compound types (classes of compound values).

Section 4 uses the generic term *accessor* to refer to struct accessors (names) and array accessors (numerical indices). It distinguishes between single-reference values that can be accessed by only one accessor and multireference values that may be accessed by more than one accessor. If there is an XSchema for the encoding, it may be possible to determine from the schema whether a value is single-reference or multireference.

The encoding rules are fairly straightforward. We will summarize the main points so you can read the actual SOAP messages in our example; for more details, see *SOAP1.2-2*. In the following summary, the `enc:` prefix is assumed to be mapped to the namespace URI of the encoding, `http://www.w3.org/2001/12/soap-encoding`.

- All values are represented by element content.
- The type of a value must be represented, either using the `xsi:type` attribute or within an agreed-upon schema.
- Within an array representation, in which all array components are represented as children of the element representing the array, the type of components can be represented just once as the value of the `enc:arrayType` attribute on the parent element.
- A simple value must have a type specified in XS2, or a type derived from an XS2 type.
- A `struct` compound value is encoded as a sequence of elements, each accessor represented by an embedded element whose name corresponds to the name of the accessor. A special appendix defines the mapping from application-defined names to XML names.
- SOAP arrays are defined as having a type of `enc:Array` or a type derived from it. SOAP arrays *must* contain an `enc:arrayType` attribute whose value specifies the type of the contained elements and the dimension(s) of the array.

To give an example, we are going to implement the Office Equipment Price service, set up TCPMon to monitor its traffic, and inspect the XML representation of Java objects in SOAP messages. Before we do that, we will quickly summarize the RPC conventions.

Representation of RPC in SOAP1.2-2

To represent an RPC, very simply, make an element whose name is the name of the remote procedure, and give it parameter children that represent the arguments. Each parameter child must have its type specified using some XML encoding, for instance, the encoding of *SOAP1.2-2*. To represent the graph structure of references, a system of `id` and `href` attributes is used: each object has an `id` attribute of type `ID` (as in *XML 1.0* DTD), and `href` attributes have, as values, the values of those `id` attributes. You will see examples in the next section.

The Office Equipment Web Service

Our implementation of this Web service will proceed through the following steps:

1. Write the Java code for the server, compile, JAR, and place into common/lib.
2. Deploy the server using a "deployment descriptor" for it.
3. Write a JSP that is a SOAP client for the service.

When we are done, we will be able to invoke the service by connecting to the JSP, and view the result shown in Figure 9-8.

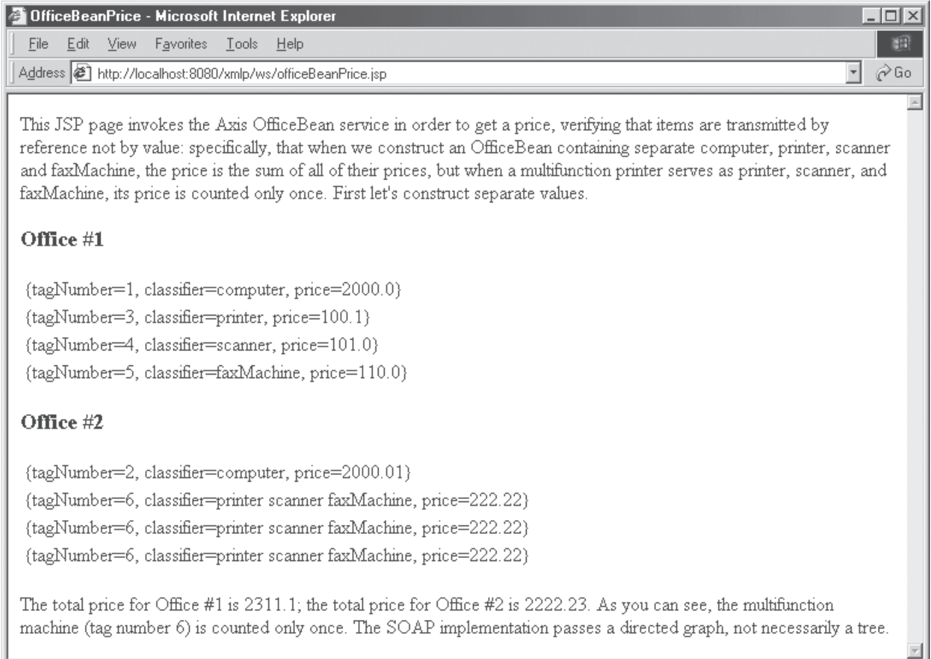

Figure 9-8. The Office Equipment Web service in operation

As the screenshot shows, SOAP correctly serializes and deserializes the graph of references.

The Java Code

The first step is to write a fair amount of very simple and repetitive Java code. We need two classes: one to represent an equipment item and the other to represent an office inventory of such items. Both classes will be Java beans. A Java bean is, very simply, a Java class that conforms to the following conventions:

- It has a default no-argument constructor.
- For each private variable that has read access, it has a get-accessor method.
- For each private variable that has write access, it has a set-accessor method.
- The accessor methods have standard signatures, and their names conform to Java bean naming conventions.

For signatures and naming conventions, it's much easier to provide an example than a definition. Listing 9-24 shows the OfficeItem class, a fully conformant Java bean.

Listing 9-24. `OfficeItem.java`*, a Java Bean*

```
public class OfficeItem{
  private int tag=0;
  private String cla="dummy";
  private double price=0.0;
  public OfficeItem(){};
  public OfficeItem(int tagNumber,String classifier,double price){
    setTagNumber(tagNumber);
    setClassifier(classifier);
    setPrice(price);
  }
// get-accessors
  public int getTagNumber(){return tag;}
  public String getClassifier(){return cla;}
  public double getPrice(){return price;}
// set-accessors
  public void setTagNumber(int tagNumber){tag=tagNumber;}
  public void setClassifier(String classifier){cla=classifier;}
  public void setPrice(double price){this.price=price;}
  public String toString(){
    StringBuffer sb=new StringBuffer("{tagNumber=");
```

```
    sb.append(tag).append(", classifier=")
      .append(cla).append(", price=")
      .append(price).append("}");
    return sb.toString();
  }
}
```

As you can see, if the name of the variable is `var` and its type is `type`, then the names and signatures of the accessors are

```
type getVar(){return var;}
void setVar(type newVarValue){var=newVarValue;}
```

The office inventory class, `OfficeBean.java`, is also a bean. (See Listing 9-25.) This time, we'll omit some of the accessor methods. To calculate the total price, we maintain a hashtable of items, and add the price of an item only if it has not yet been seen.

Listing 9-25. Another Java Bean

```
import OfficeItem;
import java.util.Hashtable;

public class OfficeBean{
  private OfficeItem[] items; // computer, printer, scanner, faxMachine
  public OfficeBean(){ // create a dummy, use four times.
    OfficeItem oi=new OfficeItem();
    items = new OfficeItem[4];
    for(int i=0;i<4;i++)items[i]=oi;
  };
  public OfficeItem getComputer(){return items[0];}
  public void setComputer(OfficeItem oi){items[0]=oi;}
// same for printer, scanner, faxMachine
  public OfficeItem[] getItems(){return items;}
  public void setItems(OfficeItem[]items){this.items=items;}
  public String toString(){
    StringBuffer sb=new StringBuffer("{");
    sb.append(getComputer().toString()).append(", ")
      .append(getPrinter().toString()).append(", ")
      .append(getScanner().toString()).append(", ")
      .append(getFaxMachine().toString()).append("}");
    return sb.toString();
  }
```

```
  private boolean isNew(Object ob,Hashtable hash){
    if(hash.get(ob)!=null)return false; // wasn't new
    hash.put(ob,ob);
    return true;  // was new, but won't be next time.
  }
  public double totalPrice(OfficeBean ob){
    double price=0.0;
    Hashtable hash=new Hashtable();
    OfficeItem[]itemList=ob.getItems();
    for(int i=0;i<4;i++)
      if(isNew(itemList[i],hash)) price+=itemList[i].getPrice();
    return price;
  }
}
```

Now we have to make this Java code into a Web service. We cannot use the previous rename-as-jws method yet. Eventually, Axis will be powerful enough to generate WSDL from our Java code and use it to define the service, but, for now, we will have to do manual compilation and deployment. This is actually useful, because manual deployment will always give you more power and flexibility than will automatic deployment.

Compile, JAR, and Copy

Before we deploy our code as a Web service, we have to make it available to the Axis engine that runs Web services. This engine is, in fact, a servlet running within Tomcat. So, we have to do the familiar sequence of operations: compile the code, put it in a JAR, and copy the JAR where Tomcat can find it, in common/lib. (We again put the sequence into a batch file, axis/OfficeBean/makeCompile.bat.) Remember that conceptually we are on the SOAP server, deploying a service.

The Deployment Descriptor and the Batch Files

The tool for manual deployment of Web services using Axis is, surprise, an XML language called *Web Services Deployment Descriptor language* (*WSDD*). Listing 9-26 is the deployment descriptor for our OfficeBean service, file deploy.wsdd.

Listing 9-26. Deployment Descriptor

```
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <service name="OfficeBean" provider="java:RPC">
  <parameter name="className" value="OfficeBean"/>
  <parameter name="methodName" value="totalPrice"/>
 </service>
  <beanMapping qname="myNS:OfficeBean" xmlns:myNS="urn:OfficeBean"
               languageSpecificType="java:OfficeBean"/>
  <beanMapping qname="myNS:OfficeItem" xmlns:myNS="urn:OfficeBean"
               languageSpecificType="java:OfficeItem"/>
</deployment>
```

This deployment descriptor defines a service and two bean mappings. The service has a name and a “provider” (RPC in this case). This means that we are using the conventions of the XML encoding and RPC as defined in *SOAP1.2-2*. The `service` element has `parameter` children. One of them specifies the class that functions as a service. The other specifies the operation that the service supports. If the service supported more operations, we would have more parameter elements, or we could use

```
<parameter name="methodName" value="*"/>
```

to indicate that all public methods of the class are operations supported by the service.

The `beanMapping` elements establish mappings between the names used in SOAP messages and Java beans that those names represent. You can find further details on WSDD in the Axis User Guide, `http://xml.apache.org/axis/index.html`.

Batch File for Deployment

The actual deployment is done by a batch file, axis/OfficeBean/deployService.bat. It’s a single-line script that we show divided into five lines. (See Listing 9-27.)

Listing 9-27. Deployment Script

```
java
-cp ../../../common/lib\axis.jar;.;../../../common/lib\clutil.jar;
../../../common/lib\log4j-core.jar;../../../common/lib\wsdl4j.jar;
../../../common/lib/xerces.jar;../../../common/lib/OfficeBean.jar
org.apache.axis.client.AdminClient -p8080 deploy.wsdd
```

The script runs a Java program, `org.apache.axis.client.AdminClient`. Before running it, we set the classpath to include a number of libraries from common/lib, including OfficeBean.jar that contains the service code. The program uses the `-p` option to set the port, and it takes one argument: the name of the deployment descriptor file. Remember to restart Tomcat after the script is run.

> **NOTE** *To run the service, you have to run this Java command (or our batch file). You only have to run it once. The information gets stored in webapps/axis/WEB-INF/server-config.wsdd. You can inspect that file and perhaps even edit it to experiment with deployment descriptors. It is read every time Tomcat is restarted.*

With the code in place and the service deployed, we can move on to write the client.

SOAP Client for the Office Equipment Service

In our PFString example, we had much of the SOAP client code automatically generated from the WSDL file. Eventually, Axis will be able to do that, but the current version (Alpha 3) is not, and so we will do some manual coding. As with manual deployment, manual coding will always provide more power and flexibility than automatic code generation.

The client, officeBeanPrice.jsp, is in webapps/xmlp/ws. (Conceptually, we are on a different machine now, the SOAP client machine.) You have seen it displayed in Figure 9-8, and it consists of these main steps:

1. Import libraries.
2. Output some explanatory text.
3. Define a few office items and two office beans.
4. Output the contents of the two offices as HTML tables.
5. Call the Web service to calculate equipment prices.
6. Output equipment prices for both offices with some explanatory text.
7. Define the `totalPrice()` method that performs the Web service invocation.

It is the last part that is truly interesting; the rest is standard JSP material. We will present all the initial parts together, with multiple omissions, then go through the Web service invocation in detail. See Listing 9-28.

Listing 9-28. The SOAP Client JSP, Part 1

```
<%@ page errorPage="../error.jsp"
  import = "org.apache.axis.AxisFault,
            org.apache.axis.client.Call,
            org.apache.axis.client.Service,
            org.apache.axis.encoding.XMLType,
            org.apache.axis.encoding.BeanSerializer,
            OfficeBean,OfficeItem,
            javax.xml.rpc.namespace.QName"
%><html><head><title>OfficeBeanPrice</title></head>
This JSP page invokes the Axis OfficeBean service...
<!-- more explanatory text -->
<% // define a few OfficeItems
  OfficeItem c1 = new OfficeItem(1,"computer",2000.00);
  OfficeItem c2 = new OfficeItem(2,"computer",2000.01);
  OfficeItem p1 = new OfficeItem(3,"printer",100.10);
  OfficeItem s1 = new OfficeItem(4,"scanner",101.00);
  OfficeItem f1 = new OfficeItem(5,"faxMachine",110.00);
  OfficeItem m2 = new OfficeItem(6,"printer scanner faxMachine",222.22);
// define two OfficeBeans, set their equipment
  OfficeBean ob1 = new OfficeBean();
  ob1.setComputer(c1); ob1.setPrinter(p1);
  ob1.setScanner(s1);  ob1.setFaxMachine(f1);
  OfficeBean ob2 = new OfficeBean();
  ob2.setComputer(c2); ob2.setPrinter(m2);
  ob2.setScanner(m2);  ob2.setFaxMachine(m2);
%>
<!-- Output the contents of the two offices as HTML tables. -->
<h3>Office #1</h3>
<table>
<% for(int i=0;i<4;i++)
     out.write("<tr><td>"+ob1.getItems()[i].toString()+"</td></tr>\n");
%>
</table>
<h3>Office #2</h3>
<table>
<% for(int i=0;i<4;i++)
     out.write("<tr><td>"+ob2.getItems()[i].toString()+"</td></tr>\n");
%>
```

```
</table><br/>
<%
try{ // invoke the service, output result with explanatory text
  %>The total price for Office #1 is <%= totalPrice(ob1) %>; the total
price for Office #2 is <%= totalPrice(ob2) %>. As you can see, the
multifunction machine (tag number 6) is counted only once...
<%
   }catch(Exception ex){
%>   Sorry, we cannot find the prices;   the exception is
<textarea rows="20" cols="80">
 <% ex.printStackTrace(new java.io.PrintWriter(out,true)); %>.
</textarea>
<%  }
%></body></html>
```

The rest of the JSP is the definition of the `totalPrice()` method. Here are the main points to watch out for as you read the code of Listing 9-29:

- The URI of the service (the "endpoint") is, as we said, a servlet that is the Axis SOAP engine.
- We create a Service and a Call object as before.
- For each class to be serialized, we create a Class object (Java meta-object for class descriptions) and a QName object to represent that class in XML.
- A QName object is, in effect, two strings: the namespace URI and the local name. The local name is the same as the name of the Java class.
- For each class, we create a Serializer object for that class.
- Once all these supporting objects are in place, we invoke the service.

Listing 9-29. The SOAP Client JSP, Part 2: Service Invocation

```
<%!
public String totalPrice(OfficeBean ob) throws Exception{
 String endPoint="http://localhost:8080/axis/servlet/AxisServlet";
  Service service = new Service();
  Call call = (Call) service.createCall();
  QName obqn  = new QName( "urn:OfficeBean", "OfficeBean" );
  Class obclass = OfficeBean.class;
  QName oiqn  = new QName( "urn:OfficeBean", "OfficeItem" );
```

```
  Class oiclass = OfficeItem.class;
// add Serializers; if we were receiving these classes as return values,
// we would add Deserializers also (commented out)
  call.addSerializer(obclass, obqn, new BeanSerializer(obclass));
  call.addSerializer(oiclass, oiqn, new BeanSerializer(oiclass));
//  call.addDeserializerFactory(obqn, obclass, BeanSerializer.getFactory());
//  call.addDeserializerFactory(oiqn,oiclass , BeanSerializer.getFactory());
  Object result;
  try {
    call.setTargetEndpointAddress( new java.net.URL(endPoint) );
    call.setProperty( Call.NAMESPACE, "OfficeBean" );
    call.setOperationName( "totalPrice" );
    call.addParameter( "arg0",  new XMLType(obqn), Call.PARAM_MODE_IN );
    result =  call.invoke( new Object[] { ob } );
  } catch (AxisFault fault) {return "Error : " + fault.toString();}
  return result.toString();
}%>
```

All the components of the Web service are now in place and you can run it by pointing your browser at `http://localhost:8080/xmlp/ws/officeBeanPrice.jsp`. However, if you want to capture and inspect the SOAP traffic, set up TCPMon to monitor it and revise the JSP to use the TCPMon's port as the service endpoint. This is what we are going to do.

The SOAP Messages

First, let us spell out what we need to do to set up TCPMon:

1. Double-click on runTCPMon.bat in webapps/axis. This will display the TCPMon administrator.

2. Fill in the `listenport` with "8081" and the `target` with "host: localhost; port: 8080".

3. Click on Add. Select the port 8081 tab.

4. Now open webapps/xmlp/ws/officeBeanPrice.jsp in a text editor and change the endpoint to be localhost:8081 instead of localhost:8080.

If you connect to officeBeanPrice.jsp again, TCPMon will capture and display both the HTTP request and the HTTP response. (TCPMon knows nothing about SOAP, it's all HTTP for it.) In fact, it will show two exchanges because the JSP

invokes the service twice, for two different offices. We will start with the first exchange (all office items are different objects), and we will show the response first because it's much simpler and very similar to our earlier examples: only a simple value is returned. (See Listing 9-30.)

Listing 9-30. SOAP Response from First Invocation

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 487
Date: Sat, 23 Feb 2002 15:02:50 GMT
Server: Apache Tomcat/4.0.1 (HTTP/1.1 Connector)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    SOAP- ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
   <ns1:totalPriceResponse xmlns:ns1="OfficeBean">
    <totalPriceResult xsi:type="xsd:double">2311.1</totalPriceResult>
   </ns1:totalPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Now the SOAP request, which is what we are really interested in. We will divide it into two parts: Listing 9-31 shows the HTTP headers and the outer elements which are the same for both requests, Listing 9-32 shows the Body element of the first request, and Listing 9-33 will show the Body element of the second request.

Listing 9-31. SOAP Request: The Skeleton

```
POST /axis/servlet/AxisServlet HTTP/1.0
Content-Length: 1676
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: "OfficeBean/totalPrice"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
<!-- the contents of the Body go here -->
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The body consists of seven elements. The first of them represents the service operation and has the same name as the method we call: `totalPrice`. The other six are all `multiRef` elements, corresponding to all the different objects at different levels of structure. Recall that we have an `OfficeBean` object that contains an array that consists of four elements, which are `OfficeItems`. They all come out as `multiRef`'s, in a somewhat random order, as shown in Listing 9-32.

Listing 9-32. SOAP Request from First Invocation: The Body

```
<SOAP-ENV:Body>
  <ns1:totalPrice xmlns:ns1="OfficeBean">
   <arg0 href="#id0"/><!-- reference to OfficeBean, id0 -->
  </ns1:totalPrice>

  <multiRef id="id0" SOAP-ENC:root="0" xsi:type="ns2:OfficeBean"
       xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:ns2="urn:OfficeBean">
   <computer href="#id1"/>
   <printer href="#id2"/>
   <scanner href="#id3"/>
   <faxMachine href="#id4"/>
   <items href="#id5"/>
  </multiRef><!-- end of multiRef for OfficeBean, id0 -->

  <multiRef id="id3" SOAP-ENC:root="0" xsi:type="ns3:OfficeItem"
       xmlns:ns3="urn:OfficeBean"
       xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <tagNumber xsi:type="xsd:int">4</tagNumber>
   <classifier xsi:type="xsd:string">scanner</classifier>
   <price xsi:type="xsd:double">101.0</price>
  </multiRef><!-- end of multiRef for Scanner OfficeItem, id3 -->

  <multiRef id="id5" SOAP-ENC:root="0" xsi:type="SOAP-ENC:Array"
        SOAP-ENC:arrayType="ns4:OfficeItem[4]"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns4="urn:OfficeBean">
   <item href="#id1"/>
   <item href="#id2"/>
```

```
    <item href="#id3"/>
    <item href="#id4"/>
  </multiRef><!-- end of multiRef for Array, id5 -->

  <multiRef id="id4" SOAP-ENC:root="0" xsi:type="ns5:OfficeItem"
        xmlns:ns5="urn:OfficeBean"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <tagNumber xsi:type="xsd:int">5</tagNumber>
    <classifier xsi:type="xsd:string">faxMachine</classifier>
    <price xsi:type="xsd:double">110.0</price>
  </multiRef><!-- end of multiRef for faxMachine OfficeItem, id5 -->

  <multiRef id="id1" SOAP-ENC:root="0" xsi:type="ns6:OfficeItem"
        xmlns:ns6="urn:OfficeBean"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <tagNumber xsi:type="xsd:int">1</tagNumber>
    <classifier xsi:type="xsd:string">computer</classifier>
    <price xsi:type="xsd:double">2000.0</price>
  </multiRef><!-- end of multiRef for Computer OfficeItem, id1 -->

  <multiRef id="id2" SOAP-ENC:root="0" xsi:type="ns7:OfficeItem"
        xmlns:ns7="urn:OfficeBean"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <tagNumber xsi:type="xsd:int">3</tagNumber>
    <classifier xsi:type="xsd:string">printer</classifier>
    <price xsi:type="xsd:double">100.1</price>
  </multiRef><!-- end of multiRef for Printer OfficeItem, id2 -->
 </SOAP-ENV:Body>
```

The SOAP request message for the second invocation is very similar in structure, except it has only four `multiRef` elements instead of six. The `multiRef` elements for office items are the same as before, and so is the `totalPrice` element. All the differences are in the `multiRef` elements for the OfficeBean and the array, of a predictable nature: there are multiple references to the same OfficeItem element. The remarkable thing, of course, is that the Java code of the SOAP server correctly interprets the XML encoding.

In Listing 9-33, we omit material repeated from Listing 9-32.

Listing 9-33. SOAP Request from the Second Invocation: The Body

```
<SOAP-ENV:Body>
   <ns1:totalPrice xmlns:ns1="OfficeBean">
      same as in Listing 9-32
    </ns1:totalPrice>
```

```
    <multiRef id="id0" SOAP-ENC:root="0" xsi:type="ns2:OfficeBean"...>
     <computer href="#id1"/>
     <faxMachine href="#id2"/>
     <printer href="#id2"/>
     <scanner href="#id2"/>
     <items href="#id3"/>
    </multiRef>

    <multiRef id="id1" SOAP-ENC:root="0" xsi:type="ns3:OfficeItem"...>
     computer OfficeItem, same as in Listing 9-32
    </multiRef>

    <multiRef id="id3" SOAP-ENC:root="0" xsi:type="SOAP-ENC:Array"...>
     <item href="#id1"/>
     <item href="#id2"/>
     <item href="#id2"/>
     <item href="#id2"/>
    </multiRef>

    <multiRef id="id2" SOAP-ENC:root="0" xsi:type="ns5:OfficeItem"...>
     <tagNumber xsi:type="xsd:int">6</tagNumber>
     <classifier xsi:type="xsd:string">printer scanner faxMachine</classifier>
     <price xsi:type="xsd:double">222.22</price>
    </multiRef>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This concludes our discussion of our last Web service example. In the code archive, we also show a "bean array" version of the prime factorization service. (The SOAP server is in `axis/PFBeanArray`; the SOAP client is invoked from xmlp/ws/pFBeanArray.jsp.) The bean array version differs from the string version presented in this chapter in that it returns an array of Java beans and therefore uses the XML encoding of *SOAP1.2-2*. It is deployed in the same way as the OfficeBean.

To conclude the entire chapter, we will return to Figure 9-1 and try to flesh it out using the material we have learned in between.

Publish-Find-Bind with UDDI

In the very beginning of the chapter, Figure 9-1 illustrated the Web services vision, and we repeat the diagram here. Figure 9-9 shows this again.

A typical scenario implied by this diagram unfolds like this:

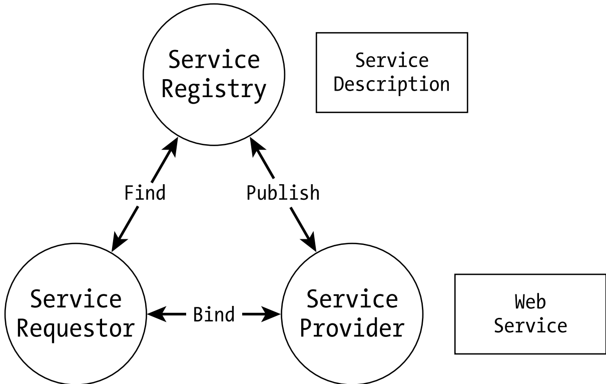

Figure 9-9. Publish, find, and bind

1. A business registers itself with a public Web services registry.

2. The business publishes Web services that it supports with a registry. This becomes public information.

3. Other businesses search the registry for services or businesses of a specific kind and find that information.

4. On the basis of this information, they connect to a Web service supported by the business, automatically or manually implement its client, and start using the service.

In this section, we will implement a simple version of this scenario, using IBM's Web Services ToolKit (WSTK), which is available at `http://www.alphaworks.ibm.com/tech/webservicestoolkit`. Be forewarned that this is a 46MB (free) archive that requires approximately 400MB of disc space to be used. It has good documentation but a somewhat complex configuration procedure. If you do decide to install it and run our example, refer to Appendix A for detailed instructions.

Scenario in Detail and in Java

Going one level closer to an implementation, we observe that, deep down inside, the registry is a database (and indeed, WSTK comes with a DB2 database in it). The database supports both authorized update access (by the business to its own business information) and public read-only access to other businesses' information. Some sort of an authorization scheme is required; WSTK implements it using an "authorization token" that is part of every restricted-access operation.

A second observation is that our registry for distributed applications is itself a distributed client-server application. We can expect that there is a client that talks to a server proxy that talks to the server that talks to the database. Our scenario, in greater detail, looks like this:

1. Create a proxy.
2. Ask the proxy for an authorization token.
3. Using the token, create an entry for a business entity.
4. Retrieve, via public access, the properties of the business entity.
5. Using the token, delete the entry from the registry.

One element of the initial scenario that we don't implement is setting up a service and creating a client for it on the basis of its WSDL description. Setting up a service would take us too far into the current details of UDDI, retrieving the WSDL is trivial, and creating a client from WSDL has been covered in an earlier section.

Java Implementation: the `main()` *Method*

We implement the more detailed scenario as a Java program, PublishFindDeleteBiz.java. To compile and run the program, click on CompilePublishFindDeleteBiz.bat in xmlp/ws/uddi; the program will pause to display its output.

Listing 9-34 shows the beginning of the class definition and the `main()` method, which follows the scenario quite closely.

Listing 9-34. The `main()` *Method of PublishFindDeleteBiz*

```
public class PublishFindDeleteBiz {
// two URLs, one for publishing (authenticated access)
// the other for public search access
static String publishURL="http://localhost:80/services/uddi/publishapi";
static String inquiryURL="http://localhost:80/services/uddi/inquiryapi";
// userid and password for creating an authentication token
static String userid="wstkDemo";
static String password="wstkPwd";
// business name: minimal info to create a business
static String businessName="N-topus Software";
```

```
public static void main(String[]args)throws Exception{
// create proxy, get authentication token, create new business
  UDDIProxy proxy=makeProxy(inquiryURL,publishURL);
  AuthToken token=getToken(proxy,userid,password);
  BusinessEntity myBiz = newBusiness(proxy,token,businessName);
// get business info: public access, no token needed
  String[][]nameKeyPairs=
    getNameKeyPairs(getBusinessInfoVector(proxy, "exactNameMatch",businessName));
// output the properties found
  for(int i=0;i<nameKeyPairs.length;i++)
    System.out.println("name="+nameKeyPairs[i][0]+
                                    "; key="+nameKeyPairs[i][1]);
// delete business using authenticated access
  deleteBusiness(proxy,token,nameKeyPairs[0][1]);
}
```

Java Implementation: Proxy, Token, and New Business

Listing 9-35 shows the first three methods called by `main()` (plus a short supporting method). They implement the first three steps in our scenario. In doing so, they rely heavily on the `org.uddi4j` library that is an essential part of WSTK.

Listing 9-35. Make Proxy; Get Authentication Token; Create Business Entry

```
public static UDDIProxy makeProxy(String inquiryURL,String publishURL)
      throws Exception{
  UDDIProxy proxy=new UDDIProxy();
  proxy.setInquiryURL(inquiryURL);
  proxy.setPublishURL(publishURL);
  return proxy;
}

public static AuthToken getToken(UDDIProxy proxy,String userid,String password)
      throws Exception{
  return proxy.get_authToken(userid,password);
}

public static BusinessEntity newBusiness(UDDIProxy proxy,
      AuthToken token, String businessName) throws Exception{
  Vector entities = unitVector(new BusinessEntity("", businessName));
  BusinessDetail bd = proxy.save_business(token.getAuthInfoString(),entities);
  return (BusinessEntity)(bd.getBusinessEntityVector().elementAt(0));
}
```

```
public static Vector unitVector(Object ob){
// supporting method: create a Vector with a single item in it
  Vector vec=new Vector();
  vec.add(ob);
  return vec;
}
```

With a business entry set up, we first demonstrate public access by retrieving its properties, and then private access, by deleting it from the registry. Note that an essential part of setting up a business entry is providing the business with an automatically generated globally unique business key. (See Listing 9-36.)

Listing 9-36. Get Properties, Delete Business

```
public static Vector getBusinessInfoVector(UDDIProxy proxy,
                  String match,String businessName) throws Exception{
// create a Vector of names to search by; one name in this case
   Vector names=unitVector(new Name(businessName));
// FindQualifiers is a UDDI4J class for search qualifiers
   FindQualifiers fQ=new FindQualifiers();
   Vector qualifier = unitVector(new FindQualifier(match));
   fQ.setFindQualifierVector(qualifier);
// find businesses by name; show only the first 99 matches
   BusinessList bList=proxy.find_business(names,null,null,null,null,fQ,99);
   return bList.getBusinessInfos().getBusinessInfoVector();
}

public static String[][] getNameKeyPairs(Vector businessInfoVec){
// for each retrieved business, store its name and key in 2-D String array of
//  business name -- business key pairs (just one pair in this program)
  String[][]pairs=new String[businessInfoVec.size()][2];
  for(int i=0;i<pairs.length;i++){
    BusinessInfo bi=(BusinessInfo)businessInfoVec.elementAt(i);
    pairs[i][0]=bi.getNameString();
    pairs[i][1]=bi.getBusinessKey();
    }
  return pairs;
} // name -- key values can be output to a stream, see main()
```

```
public static String deleteBusiness(UDDIProxy proxy,
      AuthToken token, String bizKey) {
  try{
    DispositionReport dr = // returned by every update action
      proxy.delete_business(token.getAuthInfoString(),bizKey);
    if(dr.success()) return "";
    return "Errno="+dr.getErrno()+"; ErrCode="+dr.getErrCode()+
              "\nErrInfoText="+dr.getErrInfoText();
   }catch(UDDIException ex){
     return reportUDDIException(ex); // see next section
   }catch(Exception e){
     java.io.StringWriter sw=new java.io.StringWriter();
     e.printStackTrace(new java.io.PrintWriter(sw,true));
     return sw.toString();
  }
}
```

The only remaining thing to inspect is the procedure that reports UDDI exceptions. It yields a pleasant surprise. (See Listing 9-37.)

Listing 9-37. Report UDDI Exception

```
public static String reportUDDIException(UDDIException e){
  DispositionReport dr = e.getDispositionReport();
  if (dr==null) return "";
  return "UDDIException faultCode:" + e.getFaultCode() +
  "\n operator:" + dr.getOperator() +
  "\n generic:"  + dr.getGeneric() +
  "\n errno:"    + dr.getErrno() +
  "\n errCode:"  + dr.getErrCode() +
  "\n errInfoText:" + dr.getErrInfoText();
}
```

While much of the information comes from the Disposition Report, the UDDI exception itself emits a `FaultCode`, an identifying code for the SOAP-level error condition. It turns out that a UDDI proxy is, in fact, a SOAP client, and all conversation between a proxy and the registry consists of SOAP messages. This is illustrated in Figure 9-10.

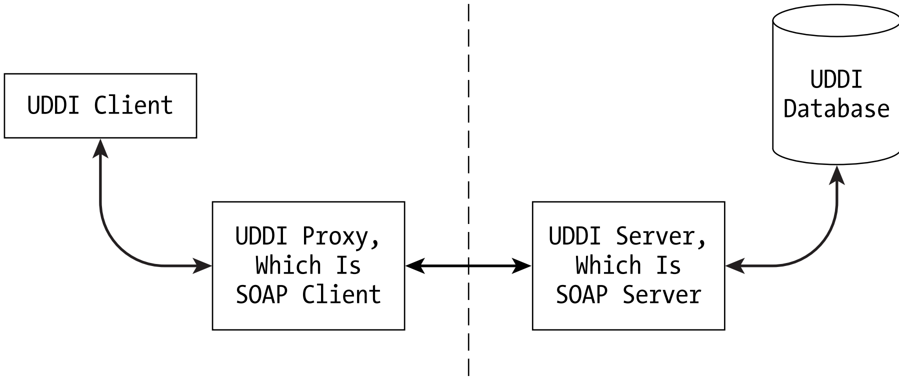

Figure 9-10. UDDI and SOAP

What this means is that we can use TCPMon to watch UDDI requests and responses. With a real UDDI registry, proxy requests directed to `publishapi` and therefore containing the authentication token would go to a secure HTTPS port, and that part of SOAP traffic would be unreadable. Because we are using a demo registry from WSTK, TCPMon can intercept and display all traffic.

SOAP Traffic Between the Proxy and the Registry

The first action involving a proxy is to obtain an authorization token. Listing 9-38 shows the request that is going out; it is addressed to `publishapi` that handles all exchanges that involve authentication.

Listing 9-38. Authorization Token Request

```
POST /services/uddi/publishapi HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 369
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<get_authToken
   cred="wstkPwd"
   generic="2.0"
```

```
    userID="wstkDemo"
    xmlns="urn:uddi-org:api_v2"/>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see, because this is a conversation between friends, the data types of arguments are not shown in the procedure call: they are presumably determined from a shared schema.

The response to this request (see Listing 9-39) is an `authToken` containing the `authInfo`, by which the registry recognizes the user. In this case, the authorization information is just the concatenation of userID with password, separated by a colon, but other registries may use other (probably more sophisticated) formulas for creating an authentication token.

Note the value of the `generic` attribute that shows the version of UDDI used: it is 2.0 in the request but 1.0 in the response. In other words, a UDDI 2.0 client is talking to a UDDI 1.0 registry.

Listing 9-39. Authorization Token Response

```
HTTP/1.1 200 OK
Server: WebSphere Application Server/4.0
Content-Type: text/xml
Content-Length: 259
Content-Language: en
Connection: close

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <authToken generic="1.0"
        xmlns="urn:uddi-org:api"
        operator="www.ibm.com/services/uddi">
      <authInfo>wstkDemo:wstkPwd</authInfo>
    </authToken>
  </Body>
</Envelope>
```

Request to create a business entry again goes to `publishapi`. The response (in Listing 9-40) starts out with providing business detail about the registry itself: its version, the namespace of its vocabulary, and its operator. (WSTK, in its responses, identifies the operator as IBM's online registry, `www.ibm.com/services/uddi`, but, in fact, the demo runs on `localhost` and does not need Internet connection at all.) The operator is reporting a `businessEntity` identified by a unique `businessKey`. Note that, in the request, the business key is

the empty string, which is a signal to the registry that a new unique key needs to be generated. Finally, there is a discoveryURL by which this business entity can be found.

Listing 9-40. Business Entity Request and Response

```
POST /services/uddi/publishapi HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 465
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<save_business generic="2.0"
   xmlns="urn:uddi-org:api_v2">
  <authInfo>wstkDemo:wstkPwd</authInfo>
  <businessEntity businessKey="">
  <name>N-topus Software</name>
  </businessEntity>
  </save_business>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
// END OF REQUEST, BEGIN RESPONSE
HTTP/1.1 200 OK
Server: WebSphere Application Server/4.0
Content-Type: text/xml
Content-Length: 583
Content-Language: en
Connection: close

<?xml version="1.0" encoding="UTF-8" ?>
  <Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
  <businessDetail generic="1.0"
     xmlns="urn:uddi-org:api"
     operator="www.ibm.com/services/uddi">
  <businessEntity businessKey="22CF7F20-28AF-11D6-BBAF-AC5F02E03C2F"
    authorizedName="wstkDemo" operator="www.ibm.com/services/uddi">
  <discoveryURLs>
  <discoveryURL useType="businessEntity"> <!-- broken into two lines -->
```

```
    http://www.ibm.com/services/uddi/uddiget?
              businessKey=22CF7F20-28AF-11D6-BBAF-AC5F02E03C2F
  </discoveryURL>
  </discoveryURLs>
  <name>N-topus Software</name>
  </businessEntity>
  </businessDetail>
  </Body>
  </Envelope>
```

The only remarkable thing about the request for information is that it is addressed to `inquiryURL`. The first line reads

```
POST /services/uddi/inquiryapi HTTP/1.0
```

The body of the request does not contain an authorization token. The response follows the familiar lines. The entire body consists of a single `businessList` element, whose start tag looks like

```
  <businessList generic="1.0"
   xmlns="urn:uddi-org:api"
   operator="www.ibm.com/services/uddi"
   truncated="false">
```

The `truncated="false"` attribute indicates that there are fewer than 99 responses, so the list didn't have to be truncated.

Finally, the request to delete the business entry goes to the `publishURL`. We will show only the `Body` element of both request and response:

```
<SOAP-ENV:Body><!-- request -->
<delete_business generic="2.0" xmlns="urn:uddi-org:api_v2">
  <authInfo>wstkDemo:wstkPwd</authInfo>
  <businessKey>22CF7F20-28AF-11D6-BBAF-AC5F02E03C2F</businessKey>
  </delete_business>
</SOAP-ENV:Body>
```

```
<Body><!-- response -->
<dispositionReport generic="1.0"
 xmlns="urn:uddi-org:api"
 operator="www.ibm.com/services/uddi">
  <result errno="0">
    <errInfo errCode="E_success">E_success (0) Success. </errInfo>
  </result>
</dispositionReport>
</Body>
```

The entire response body is a disposition report that in this case reports success. On this hopeful note, we can conclude the UDDI section and the entire chapter. Just as with WSDL, we believe that something like a UDDI registry will be an important part of the emerging web of distributed applications, and we are fairly confident that the operation of such a registry will be built on top of the same XML protocol that the applications themselves are using. We also believe that, when the registry specification converges to a standard, it will be substantially different from today's UDDI.

Conclusion

In this long chapter, we have covered the emerging field of Web services. We have tried to separate the vision and promise of Web services from their current reality and the frequent exaggerations of that reality. We have also tried to separate relatively stable features from multiple details that are still in flux and await standardization.

The first section of the chapter introduced the general concept of a Web service and its main building blocks. A review of relevant specifications—SOAP, WSDL, and UDDI—served both to elaborate the building blocks structure and to emphasize how incomplete and unfinished those specifications are. Although SOAP has a considerable base of users and is on track for standardization by W3C, both WSDL and UDDI are best described as placeholders for eventual standards that will perform the same function but may be very different from the current drafts.

The second section of the chapter traced all aspects of a Web service development within the context of a simple example. That example used the Axis framework for both the SOAP server and SOAP client. The next section developed alternative clients outside the Axis framework and in different languages to illustrate the main promise of Web services: complete interoperability between different languages and platforms.

After the examples, the next two sections presented an overview of *SOAP 1.2*, both its core features (*SOAP 1.2*, Part 1) and Adjuncts (*SOAP 1.2*, Part 2). Within Adjuncts, we particularly concentrated on SOAP XML encoding and SOAP conventions for representing RPCs. To illustrate this material, we developed, in the following section, another example that uses many more features of XML encoding than did the first example.

Finally, in the last section, we showed how to create a business entry in a UDDI registry, using IBM's WSTK. We also showed how to query the registry and how to modify the entry (in a rather drastic way, by deleting it). At this point, we are all set to add more content to the entry, including search terms by which we want our business to be found and a WSDL description of its services. (The search terms would include standard business categorization codes, as specified in NAICS, UN/SPC, and other standards.)

With this content in place, the Web services offered by the business can indeed be discovered by a software agent, and clients for the services can be automatically generated from their WSDL descriptions. However—and we do want to insert a word of caution—even with all this machinery in place, we will still be quite a distance away from the vision of software agents creating long chains of Web service invocations to create solutions to complex problems. Composing simple Web services into a complex meaningful whole requires a leap in intelligence to which the current tools do not even begin to aspire. Despite such caution, we can honestly express enthusiasm: there are many good things to be done, and, although some of them ought to be developed as open-source infrastructure, there's also money to be made.

APPENDIX A

Installation Guide

THIS BOOK'S CODE falls into three categories: Java-JSP code running on the Java platform, VBScript-VB-ASP code running on the Windows platform, and XML-XSLT code running on both platforms. To run and experiment with all of our code, you have to configure two platforms. This book can be usefully read with just the Java-based code for the simple reason that there is much more Java XML code than there is Windows XML code (unless you include the .NET platform, which is not covered in this book). Just as important, the Java XML code is largely open source: this helps in debugging and in learning from the code. We want to emphasize that much of our own Java code forms a testbed for using XSLT and other cross-platform technologies, and it is possible to use the testbed without active understanding of all the details of its code. We do provide Windows-based alternatives for many Java XML tools.

> **NOTE** *Both in this appendix and throughout the book, we assume that you are familiar with HTML and the basics of Web applications: the HTTP protocol, CGI, and its more recent alternatives (ASP, servlets, and JSP). These are covered in Appendix B and C.*

On both platforms, you need the following types of software:

- Web server with backend connector (ASP or JSP)
- XML parser that is namespace aware and XML-Schema capable
- XSLT processor
- Bean Scripting Framework (BSF) for running scripts from within XSLT stylesheets
- RELAX NG validator and accompanying software
- Web services toolkit: SOAP and WSDL

BSF is not needed until Chapter 6, RELAX NG software is not needed until Chapter 8, and Web services software is not needed until Chapter 9. However, you'll need a Web server or two, an XML parser, and an XSLT processor from the beginning.

In the case of the Java platform, you also need to install the platform itself.

Version Updates

This appendix describes the versions of software and their locations as of the time the book was finished (March 2002). New versions are unavoidable, and their locations may change. For updates, please go to the top-level readme.htm file in our code archive, which is downloadable from the book's Web site at `http://www.apress.com/catalog/book/1590590031/`. We will try to keep it up to date.

As mentioned in the introduction, there will be a mailing list for readers of the book where you can post questions. Instructions for joining the list can be found in the same readme.htm file.

The Java Framework

To install the Java framework, proceed as follows:

1. Install Java 2 Standard Edition from `http://java.sun.com/j2se/`. The archive you will actually download is known as the *Java Development Kit* (*JDK*). As of this writing (March 2002), the current version is jdk1.4, but we used version 1.3.1 for the book's code. Version 1.4 includes some libraries that we had to download separately for the UDDI section of Chapter 9.

   After the JDK is installed, add the JDK bin directory to your environment PATH variable.

2. Install Apache Tomcat, a combined Web server and JSP processor, from `http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/`. In that directory are subdirectories for different versions, and within each version subdirectory is a bin subdirectory that contains the actual binary distributions. We have used version 4.0.1, whose distributions are in the v4.0.1/bin/ subdirectory of the release directory.

   There are three possibilities for download: zip and gzip archives and a Windows-specific EXE installer. Download any of these three and

uncompress into a directory of your choice. In our own installation, the name of that directory is tomcat401, at the top level. We will refer to that directory as TOMCAT_HOME. TOMCAT_HOME is the centerpiece of the framework; the rest of the Java framework and our own code will go into subdirectories of the Tomcat home directory.

> **NOTE** *If you are installing on a Windows NT or 2000 machine and are using the EXE installer, you will have the option of installing Tomcat as a service.* Do not *select this option because the console (command line) window in which Tomcat is started (if it is not a service) may show valuable error messages. Otherwise, download the installer, run it, and choose the installation directory as just described.*

3. To start and stop Tomcat, use the scripts in the TOMCAT_HOME/bin directory. The scripts are startup.bat and shutdown.bat for Windows, and startup.sh and shutdown.sh for Unix. Once Tomcat has started, point your browser to `http://localhost:8080` and run the included servlets and JSP examples to test the installation. Additional instructions are available from the Apache Web site at `http://jakarta.apache.org/tomcat/tomcat-4.0-doc/RUNNING.txt`; when you have Tomcat running, the same page is available locally at `http://localhost:8080/tomcat-docs/RUNNING.txt`. (We use "/" generically for either "/" or "\".)

4. Download Xalan-J 2.2 (or later) XSLT processor from `http://xml.apache.org/dist/xalan-j/`. Make sure that you download the latest stable version. Note that there are zip and tar.gz archives of the same package; the .tar.gz archive is smaller.

   Unzip into a directory of your choice. Move all the JAR files from that directory into TOMCAT_HOME /common/lib. Both among those JAR files and in Tomcat's common/lib directory there will be a xerces.jar (the XML parser). Keep the later of the two versions.

5. For Chapter 7 and later, repeat the same process with Rhino JavaScript from `http://www.mozilla.org/rhino/download.html`: download the most current stable release, unzip, and move the JAR file into TOMCAT_HOME /common/lib.

6. For Chapter 9, repeat the same process with the Axis Alpha 3 SOAP toolkit from `http://xml.apache.org/axis/index.html`: download, unzip, and move the JAR files into TOMCAT_HOME/common/lib.

The distribution contains a webapps directory with an axis subdirectory; copy the axis subdirectory into TOMCAT_HOME/ webapps. Additional installation instructions for the examples in Chapter 9 can be found in a separate section later in this appendix.

7. For Chapter 8, download RELAX NG validator Jing and the RELAX NG implementation of XHTML modularization from `http://www.thaiopensource.com/relaxng/`. In more detail, abbreviating this URL to `ThaiHome`, proceed as follows:

   Download jing.jar and, if on Windows, the Win32 executable, jing.exe, from ThaiHome/jing. Place jing.jar into TOMCAT_HOME /common/lib. Place the Win32 executable on your Windows path.

   Download the XHTML modularization package as ThaiHome/xhtml/xhtml-rng.zip from ThaiHome/xhtml/. Unzip in a directory of your choice: we do not use the package in our programs but use its materials as examples of good design.

8. Finally, unzip our own code, xmlp.zip, into TOMCAT_HOME/webapps/. (The webapps subdirectory is created within Tomcat at installation; this will create the xmlp directory within TOMCAT_HOME/webapps. It will also add some files to the axis directory within TOMCAT_HOME/webapps.)

   One JAR file (TOMCAT_HOME/webapps/xmlp/WEB-INF/classes/XslUtil.jar) in our code needs to be moved to TOMCAT_HOME/common/lib. In the same WEB-INF/classes directory is the source file, XslUtils.java, and a Windows batch file, make.bat, in case you decide to edit the code: running make.bat will recompile the class, re-JAR it, and recopy the jar to TOMCAT_HOME /common/lib.

   Note that we provide command line scripts only for Windows, as BAT files, assuming that Linux users will have no difficulty recasting them as Unix shell scripts.

   If Tomcat was running when you installed our code or when you moved XslUtil.jar to common/lib, stop and restart it before continuing.

You can point your browser to `http://localhost:8080/xmlp`, and you will be looking at our default file (index.html) that contains links to activate or display our examples. Some links result in examples being run, whereas others simply

display the code. (The text of the link indicates what action to expect.) At this point, you can test the examples of Chapter 1; later chapters require additional installations as explained in the remaining sections of this appendix.

The Windows Framework

On Windows, the framework is mostly part of the platform. You have to install an IIS or PWS Web server and make sure that it is running. (A common arrangement is to have it start at computer startup.) By default, the Windows Web server runs on port 80 and Tomcat runs on port 8080. Both can be in operation at the same time, and each can refer to the other's pages with no difficulty.

The Server

Most versions of IIS/PWS currently in operation support ASP, but please note that IIS 3.0 as it is installed by default with Windows NT 4 Server does not. Upgrading IIS 3.0 to IIS 4.0 on Windows NT is an extremely painful procedure. If you are in this category of users, we recommend that you either upgrade to Windows 2000 or forego testing the ASP examples.

The Parser and XSLT Processor

For XML parser and XSLT processor, we use MSXML3. This is included in IE6, so, if you have IE6 installed, you don't have to do anything. For IE5.0 and IE5.5, the installer (msxml3sp2Setup.exe) can be downloaded or run directly from this URL (shown as two lines):

```
http://download.microsoft.com/download/xml/
   SP/3.20/W9X2KMeXP/EN-US/msxml3sp2Setup.exe
```

This is the English version, Service Pack 2 (SP2). The referring page for SP2 in general (again shown as two lines) is

```
http://msdn.microsoft.com/downloads/sample.asp?
   url=/msdn-files/027/001/772/msdncompositedoc.xml&frame=true
```

On this page you will find versions for other languages, as well as three "Important Notes". The first one states that, if you have IE6, you don't need to upgrade, and the second states that this upgrade will install only in "replace mode" not in "side by side mode." What this means is that, with an earlier

upgrade, you could choose to install MSXML3 side by side with MSXML2, but this option is no longer available (and you don't want it, anyway: MSXML is infamous for supporting a nonstandard version of XSLT). The third Important Note says that, if you upgrade IE5.0, all applications using MSXML must be closed. As usual, it's best to close all applications.

Note on MSXML4

This is a warning: MSXML4 is not a substitute for MSXML3 precisely because it will *not* install in a replace mode. In other words, if you install MSXML4 with IE5.0 or IE5.5 on your machine, the browser will still be running the old, substandard MSXML 2. MSXML4 is good for writing server-side code that uses XML/XSLT, but it will not help you at all with browser-based XML/XSLT code.

Even on the server, MSXML4 may be short-lived because .NET doesn't use it at all.

The SOAP Toolkit

Optionally, you may wish to use the Microsoft SOAP Toolkit to experiment with the SOAP examples of Chapter 9. The toolkit can be found at the following URL, which is shown divided into two lines:

```
http://msdn.microsoft.com/downloads/default.asp?
  URL=/code/sample.asp?url=/msdn-files/027/001/580/msdncompositedoc.xml
```

The Final Steps

Unzip our code, xmlpasp.zip, into the root directory of the drive that has the IIS Web server on it; it will unzip into \inetpub\wwwroot\xmlp.

Your Windows installation is complete, except for two details:

- Listing 3-1, if run over HTTP, requires very relaxed security that enables ActiveX controls in the page. Running this from localhost may require adding localhost to your "trusted sites." Even this may not always work, or you may be unwilling to change security settings even temporarily. If this is the case, simply run this example as a local file; that is, open TestValidityJS.htm in the browser as a local file. The intent of the example is demonstrated even better if it is run as a local file.

- One program in Chapter 4 requires installing a DLL, as explained in the next section.

Installing a DLL

Although most Windows programs in the book use VBScript or JScript within an ASP, ActiveSAXbookpicker from Chapter 4 is an ActiveX control. Installing an ActiveX control is simple, but there is a preliminary step if you don't already have any DLLs based on Visual Basic 6.0. DLLs compiled with VB6.0 require the file MSVBVM60.DLL in a standard DLL directory such as WINDOWS\SYSTEM32. If you don't have this file, it is available from many Web sites (search Google with "download" and "msvbvm60.dll" as search terms; it's a common problem) or you can run an installer (URL divided into three lines):

```
http://support.microsoft.com/default.aspx?
  scid=http://download.microsoft.com/download/vb60pro/
    Redist/sp5/WIN98Me/EN-US/VBRun60sp5.exe
```

With this proviso, the installation instructions are as follows:

- Make sure that you have MSVBVM60.DLL in a DLL directory.
- Move ActiveSAXbookpicker.dll from TOMCAT_HOME/webapps/xmlp/ActiveSAX_VBCode/ActiveSAXBookpicker into the same directory.
- Register the ActiveSAXbookpicker.dll as an ActiveX server.

The last task is performed with the following command:

```
REGSVR32 <pathToDLL>
```

The easiest way to do this without typing errors is to select Start ➢ Run, type `regsvr32` into the command line followed by a space, and then drag the ActiveSAXbookpicker.dll file icon over to the Run box.

Note to Users of Older Versions of Windows

You don't need to upgrade to Windows 2000 or XP, except for installing WSTK 3.0 for the last example of the last chapter (if you do run that—see the "UDDI Example" section later in this appendix). All code has been tested on Windows 98, and we've noted two aggravations. First, when running more stressful examples, you are more likely to have an actual system crash than when running with a more stable operating system; for this, there's nothing you can do but reboot. Second, you'll find that, when running some of our batch files, you will see an

"Out of Environment Space" error message. To get around this, make a shortcut to the batch file and right-click on the shortcut's icon. Choose Properties ➢ Memory and replace the Auto selections with the highest available numerical values (probably 640 on the left and 4096 on the right). Now double-click on that shortcut whenever we ask you to double-click on the batch file itself, and it should work.

If Space Is at a Premium

All together, the archives you have downloaded will occupy more than 50MB of space, including more than 40MB for the Java platform and language, jdk.1.3. Most of that space is consumed by documentation and source code (for open-source software). If you want to minimize your space expenditures, do this: after downloading the Xalan and Axis archives, extract only the executable (JAR) files and get rid of the archive itself. This will reduce your space commitment significantly. You can still view, for example, Xalan documentation (which includes Xerces documentation) on the Web at `http://xml.apache.org/xalan-j/index.html`. For more on online resources, see Appendix D.

To reduce space requirements for the UDDI example of Chapter 9, use an external UDDI registry, as explained in the "UDDI Example" section later in this appendix.

If space is not a problem, however, we highly recommend that you download not only the binaries with documentation and samples, but also the source files for these distributions. This will give you examples to learn from, code snippets to copy and modify, and source to go along with line-numbered error messages in debugging.

Database Connectivity

The application of Chapter 7 uses a relational database to store XML data. This section of the appendix describes what needs to be done to get that application running. Because the application is written in Java, we consider only two cases: Java platform on top of Windows and Java platform on top of Linux.

Java Platform on Windows

We supply an initial almost-empty Access database that the user can further populate and modify. The database, xmlp.mdb, is in our xmlp.zip archive, within the TOMCAT_HOME/xmlp/dat directory. To make it available to the Java application, go into the ODBC control on your system and create an ODBC Data Source Name (DSN), as explained in the next two paragraphs.

On Windows 2000 systems, the ODBC control is in Administrative Tools in the control panel. Click on Data Sources (ODBC) to open a tabbed dialog box. Under the System DSN tab, click on Add, and then on Microsoft Access Driver. In the next dialog box, enter "xmlp" as the name of the data source, then click on the Select button and enter "TOMCAT_HOME/xmlp/dat/xmlp.mdb" as the data source (or navigate to it to select).

In Windows NT 4, start the ODBC applet in the control panel. Under the System DSN tabbed panel, click on Add, select Microsoft Access Driver, and then type "xmlp" as the name of the data source, and click on Select to navigate to our file (xmlp.mdb) in TOMCAT_HOME/webapps/xmlp/dat. Type any useful comment in the Description box and click on OK.

Java Platform on Linux

For Linux installation instructions follow the link to linux_readme.htm in the top-level readme.htm file of the code archive.

Large Data Files

Several of our programs use large data files that are not included in our archive. In particular, we use the King James Bible with XML markup by Jon Bosak (`http://www.ibiblio.org/bosak/`), and Shakespeare's plays from the same source. The Bible (together with the Koran and the Book of Mormon) can be found at `http://www.metalab.unc.edu/bosak/xml/eg/rel200.zip`. Please download (almost 2MB zipped), extract ot.xml and nt.xml, and move into TOMCAT_HOME/webapps/xmlp/dat/jb/. The complete collection of Shakespeare's plays is in shaks200.zip in the same directory, but we include *Macbeth* and *Julius Caesar* (macbeth.xml, j_caesar.xml) in our xmlp.zip file so that you can test our code without another 2MB download.

Some examples in Chapter 6 use a small excerpt (included in our code archive) from the CIA World Factbook. Those readers who are interested in the entire Factbook database may download it from `http://www.cia.gov/cia/publications/factbook/index.html`.

Web Services Examples (Chapter 9)

As mentioned in the "The Java Framework" section, you have to download and install Axis to run Web services examples. Follow the instructions in that section before proceeding. As part of installing and running the examples, you will need to run short scripts, which we again provide only for Windows (as BAT files). In that sense, but only in that sense, the instructions are Windows-specific.

Chapter 9 has three examples that require additional comment: PrimeFactorsString, OfficeBean, and UDDI.

PrimeFactorsString

In this example and the next, some Java code needs to be compiled, put in a JAR file and copied to the common/lib directory. The JAR file for this example, PFString.jar, is already in our archive, in the webapps/axis/PFString/ directory. If your Axis version is the same we are using (Alpha 3), then simply copy that file to common/lib. If you have a later version, you may want to recompile the example, in case there are incompatibilities between versions. Proceed as follows.

The first step is to double-click on the file webapps/axis/PFStringW2J.bat. This generates Java client files in the webapps/axis/PFString/localhost subdirectory. Next, double-click on webapps/axis/makePFString.bat. This will compile the Java files, put them into PFString.jar, and copy the JAR to common/lib.

After PFString.jar is copied to common/lib, you have to restart Tomcat.

OfficeBean

In this example, you have to compile and JAR the code before copying it to common/lib. You also have to "deploy" the service.

To do this, double-click the file webapps/axis/OfficeBean/makeCompile.bat. This compiles the source code in that directory, puts it into OfficeBean.jar, and copies the jar to common/lib, where Tomcat will find it on its next restart.

Next, double-click on the file webapps/axis/OfficeBean/deployService.bat. This will deploy the service, effective immediately (no further restart is needed).

UDDI Example

The UDDI example requires an installation that is much more involved than for any other example. Instructions are provided in a readme.htm file for the UDDI example in our code archive, which is reachable from the main index.html page.

Additional Platforms

Many of our readers will want to do at least some of their work with Perl, PHP, C++, C#, or other platforms. XML toolsets are available for all of them, and most of this book will still be relevant. Although we can't tell you how to translate each example into your target framework, we do hope to hear from you as you find and solve the difficulties involved.

APPENDIX B

Web Applications

A Web application is a backend program run by a Web server that receives input data from a Web client and sends the output of the program back to the client. As explained in Appendix C, input data comes from the client either as a query string attached to the request URI (with the GET method) or as the body of the request (with the POST method). Several frameworks establish communication between the Web server and the backend program, and we consider four of them in this appendix: Common Gateway Interface (CGI), Active Server Pages (ASPs), servlets, and Java Server Pages (JSPs).

General Framework

Consider an HTML form, as shown in Listing B-1.

Listing B-1. HTML Form with GET Method

```
<form action="http://localhost:8080/xmlp/nobody.jsp" method="GET">
  <input type="text" name="alpha" value="1"/>
  <input type="text" name="beta" value="2"/>
  <input type="Submit"/>
</form>
```

When you click on the Submit button, the browser sends the request shown in Listing B-2 to the server specified in the `action` attribute (in this case, `localhost` port 8080). We have deleted a few headers for simplicity.

Listing B-2. Request Corresponding to Listing B-1

```
GET /xmlp/nobody.jsp?alpha=1&beta=2 HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Host: localhost:8080
Connection: Keep-Alive
```

It is then up to the server at `localhost:8080` to decide what `/xmlp/nobody.jsp?alpha=1&beta=2` is and what to do with it. The important thing to realize is that the server is getting exactly what it would get if you had typed `http://localhost:8080/xmlp/nobody.jsp?alpha=1&beta=2` into the address window of your browser.

If you use the form of Listing B-1 but replace GET with POST, your request will be as shown in Listing B-3.

Listing B-3. Request Corresponding to HTML Form with POST Method

```
POST /xmlp/nobody.jsp HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Host: localhost:8080
Content-Length: 14
Connection: Keep-Alive

alpha=1&beta=2
```

As you can see, the data is now separated from the headers by an empty line. There is a `Content-Type` header, which says how the data is formatted, using a MIME data type. Finally, there is a `Content-Length` header, which says how much data there is.

Different frameworks differ in how they invoke the backend program to process the data, and how the data is presented to the programmer who is writing the backend program.

CGI

CGI is by far the oldest of the frameworks, going back to the early 1990s. A traditional CGI program is, indeed, a separate self-standing program, running in a different process and invoked by the server using the underlying operating system. (By contrast, modern backend frameworks are run by the server itself, as a thread within the same process, which requires much less time and other computational resources.) The CGI program communicates with the server via system-level environment variables, which include `REQUEST_METHOD`, `QUERY_STRING`, `CONTENT_LENGTH`, and `CONTENT-TYPE`. If the method (that is, the value of the `REQUEST_METHOD` variable) is `GET`, the CGI program would parse the value of the `QUERY-STRING` variable, extracting names and values of arguments. If the method is POST, the CGI program would read the request's data from standard input, ending when it has processed the `Content-Length` number of bytes or eventually timing out if it couldn't. In either case, it would write the resulting page, with appropriate `Content-Type` and `Content-Length` headers, to standard output. Note that the programmer has to output literally everything in the request: the headers, the blank line, and the entire HTML page, with tags and all.

There is more, but not much more than that to CGI. For further information, go to `http://www.w3.org/CGI/`, from which you'll be redirected to various documents at the National Center for Supercomputer Applications (NCSA) at the

University of Illinois Urbana-Champaign (UIUC), which is where the original Mosaic browser, HTTPd server, and the CGI protocol were invented. You may want to start with `http://hoohoo.ncsa.uiuc.edu/cgi/intro.html`.

Improvements to Backend Processing

Improvements to backend processing since the initial CGI days fall into four main categories:

- Backend programs are run by the server itself, as a thread within the same process, without invoking the operating system. This makes it possible to keep backend programs in memory between invocations, thus greatly improving performance.

- Backend programs can maintain state, "remembering" the identity of the user between invocations. This is essential for extended interactions, such as e-commerce. A logical sequence of interactions with the same user is called a *session*.

- Processing of request and response is wrapped into object-oriented interfaces that hide the tedious detail from the programmer.

- Backend code can be embedded into an HTML page within special tags, so the HTML page serves as a template into which computed data is inserted in specified places.

Microsoft's Active Server Pages (ASPs) show all these facilities, except the languages they use (JScript and VBScript) are not really object-oriented programming languages. This will change in ASP.NET.

ASPs

An ASP provides a Request object, a Response object, and a Session object. The Request object knows how to read request data; the same methods extract data from the query string and from the request body. The Response object knows how to write data to the client. The Session object is, in effect, a table for objects that may be referenced by the same user in an extended, multirequest interaction; the Session object holds data that persists through all requests from the same user within the same session. In addition to the Session object, ASP provides an Application object to hold data that persists through all sessions,

throughout the lifetime of the application. As of ASP 3.0, ASP also provides an ASPError object that knows what has just gone wrong.

For a simple example (such as Listing B-4), suppose that we want a session to begin with a username specification to which we can refer later. Then page hello.asp does that. Note that the programmer doesn't have to send tags.

Listing B-4. ASP with Request and Session Objects

```
<%@ LANGUAGE="VBSCRIPT" %>
<% Option Explicit %>
<html><head><title>Hello </title></head><body>
<%
DIM username
username = Request.QueryString("user")
IF(""=username) THEN
 username=Session("username")
ELSE
 Session("username")=username
END IF
%>
<H1>Hello, <%= username %>!</H1>
</body></html>
```

Note that the ASP page can have both template HTML and snippets of programming code. In fact, any legal HTML page is also an ASP, so you can change its extension to .asp and it will display correctly. The only difference is that it will be passed through an ASP processor (because that's where the server sends all URL requests with the .asp extension).

The response to `http://localhost/xmlp/hello.asp?user=Joe` will be:

```
<html><head><title>Hello </title></head><body>
<H1>Hello, Joe!</H1>
</body></html>
```

and, if we later look at `http://localhost/xmlp/hello.asp` with no username specified, the value is remembered, and the page is the same. Using a TCP monitor tool, TCPMon (introduced in Chapter 9), we can track the HTTP requests and responses, and establish that ASP maintains sessions using *cookies*, small and dated text files that the server can place on the client's computer and receive back as a header in the next request within a session. A cookie header looks like this (from running the example of Listing B-4):

```
Cookie: ASPSESSIONIDFFFOMNIT=IONLHGCDBBNCBJOFIHJGMPGD
```

The Request object receives the cookies just as it receives all other headers and collects them in its `Request.Cookies` property. The Response object knows how to write cookies; we can refer to `Response.Cookies(0)` or to `Response.Cookies("cookieName")`.

Plenty of books have been written on ASP programming, and you can find good tutorials on the Web. See Appendix D for some of them.

One last detail about ASPs is that switching from GET to POST means switching from the `Request.QueryString` object to a `Request.Form` object, which is simple and logical, but mildly annoying because you may well use forms with `method="GET"` for debugging (so you can see the data in the URL) and then switch to `method="POST"` to hide the data. This is better done in Java servlets.

Java Servlets and JSPs

Java servlets are, in some respects, a step forward from ASPs because they use a real programming language with real objects. In other respects, they were a step back, because they are Java programs that are not embedded into an HTML (or XML) template, and so the programmer has to output all the tags manually, and even some of the headers. The equivalent of the ASP page of Listing B-4 is the `HelloServlet.java` class of Listing B-5. It imports every class to be used, then defines `doGet()` and `doPost()` methods to handle the different invocations. The `doPost()` method simply calls `doGet()`.

Listing B-5. Sample of Servlet Code

```
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
  String username=request.getParameter("user");
  HttpSession session = request.getSession(true);
  if(null==username)
    username=(String) session.getAttribute("username");
  else session.setAttribute("username", username);

  response.setContentType("text/html");
```

```
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Hello World!></title></head>");
    out.println("<body><h1>Hello,");
    out.println(username);
    out.println("!</h1></body></html>");
  }

  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
    doGet(request, response);
  }
}
```

This is invoked as `http://localhost:8080/servlet/HelloServlet?user=Joe`. Servlets can be useful, but we don't use explicitly written servlets in this book; instead, we use JSPs. A JSP is a template page with special tags for code that is intended to become a servlet. When a JSP is accessed for the first time, it is rewritten as a servlet and compiled, which is why the first invocation takes a bit longer. Otherwise, a JSP looks rather like ASP with some extra flavorings, at least at first sight. The JSP of Listing B-6 is equivalent to the ASP of Listing B-4.

Listing B-6. JSP with Request and Session Objects

```
<%@ page errorPage="error.jsp" import="java.util.Properties"
%><jsp:useBean id="dict"  class="java.util.Properties" scope="session"
/>
<html><head><title>Hello </title></head><body>
<%
  String username=request.getParameter("user"); <!– same for GET and POST –>
  if(null==username)
// get user name from dict using "username" as key;
// if not found, use "dear User" as default
    username = dict.getProperty("username","dear User");
  else dict.setProperty("username",username);
%>
<h1>Hello, <%= username %>!</h1>
</body></html>
```

The fundamental difference from ASP is that this is not a script to be executed, but a shorthand notation for a program to be compiled (and then executed); if this is the file /webapps/xmlp/hello.jsp, it will be rewritten as hello$jsp.java in TOMCAT_HOME/work/localhost/xmlp/, which will be compiled into TOMCAT_HOME/work/localhost/xmlp/hello$jsp.class. The compilation improves performance, but its primary benefit is error checking.

The Java compiler can catch lots of simple mistakes because Java—unlike VBScript or JScript—is a strongly typed language, so the compiler can ensure that there are no type errors or missing declarations in the code.

Many excellent resources on JSPs are available, starting from `java.sun.com/jsp`. In this appendix, we will limit ourselves to a few comments on Listing B-6. It starts by specifying the error page; this is simply another JSP page that will be presented to the user if an unhandled error condition occurs. It can be as simple as that shown in Listing B-7.

Listing B-7. A Simple Error Page

```
<%@ page isErrorPage="true" %>
<html><head><title>ErrorPage</title></head><body>
Problem: <%= exception.getMessage() %>
</body></html>
```

We will discuss the use of error pages in Appendix E "Troubleshooting in JSP".

After the error page, we tell the JSP processor to import the `java.util.Properties` class to compile this code. Then we declare a bean, a Properties object named `dict`, whose scope is this session. In other words, we are creating a dictionary that is kept by some sort of Session object, but we don't have to think about the Session object: we simply say that we want our dictionary to be available throughout the session and to go away once the session is over. Objects can have one of four possible scopes: page, request, session, or application, and with each scope they will last for as long as they are supposed to. (Page scope is that of an ordinary variable declared on the JSP page; a request-scope object will follow the request to another JSP or servlet if the request is forwarded.)

ASP.NET will undoubtedly catch up on some of the advantages of JSP over ASP. Looming behind both are J2EE and the entire .NET framework. Both are fascinating developments that place Web applications in a larger context of distributed applications with persistent data. Unfortunately, both are also outside the scope of this book.

APPENDIX C

HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is the communication protocol of the Web. It was invented by Tim Berners-Lee as part of the Web's suite of specifications that also included definitions of HTML and URL. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0 became an Internet standard, released by the Internet Engineering Task Force (IETF) as RFC 1945. HTTP 1.0 improved the protocol by allowing messages to be in a MIME-like format, containing meta-information about the data transferred and modifiers on the request/response semantics. The current version, HTTP/1.1, made performance improvements by making all connections persistent and supporting absolute URLs in requests.

HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80, but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport, and any protocol that provides such guarantees can be used.

This appendix is not intended as a thorough description of HTTP. We give only as much detail as is needed for understanding the book's explanations and examples.

URIs, URLs, and URNs

A Uniform Resource Identifier (URI) is a means of unambiguously referring to a resource. The resource can, in principle, be anywhere, but typically URIs refer to resources on the Internet. The URI specification is an Internet standard (RFC 2396). Resources located by URIs can be files, email addresses, programs, services, or even books in the library. URIs are of two types: Uniform Resource Locators (URLs) and Uniform Resource Names (URNs). URLs are more familiar, but W3C defines *URL* as "an informal term (no longer used in technical specifications) associated with popular URI schemes: http, ftp, mailto, etc." (`www.w3.org/Addressing/`).

As you can see from this quote, the first component of a URI is called its *scheme*. The scheme of URNs is `urn`. The scheme of HTTP resources is `http`; in general for URLs, the scheme is the name of the protocol that is used to access the server. An HTTP URN (or URL) is a pointer to a particular resource on the

Internet *at a particular location*, such as `www.apress.com/catalog/book/1590590007/index.html`. A complete URL must specify the scheme (that is, the protocol used to access the server), the name of the server, the directory path from the server root to the resource, and the name of the resource; optionally, it can specify the port number. If the port number is left out, the default port (80 for HTTP) is assumed. Many browsers allow the user to omit the scheme (taking `http` as the default), and, for HTTP resources, also omit the string `www.` in the beginning of the server name.

URNs are intended to serve as persistent, location-independent resource identifiers. Given a URN, a client will be able to retrieve it from any server that has the resource. For URNs to function that way, there must be an agreed-upon system of registries that would map URNs to locations. HTTP exclusively deals with URLs.

As this book explains (especially in Chapter 2), both URNs and URLs are often used in the XML world as simply unique identifiers of namespaces, without any meaning attached to them (such as "this is an address of a resource on the Web"). Within the SOAP world, they are also used as unique identifiers of various things, such as XML encodings. Many people feel that this is a perversion of the original intent of URLs, even if it is approved of by the URL's original inventor.

An HTTP URL can be followed by a location within the resource, separated from the resource URL by the # character. In HTML resources, the string that follows must be the same as the `ID` or `NAME` attribute of an `A` element within the document. In XML, the string that follows is an XPointer, as explained in this book.

An HTTP URL can also be followed by a query string, separated from the URL by the "?" character. A query string is a list of arguments to a backend program (CGI script, ASP, servlet, JSP, and so on—see Appendix B). The arguments are separated by an ampersand ("&"), and each argument is of the form `name=value`. The arguments (and the rest of the URL) must be URL encoded: a character that is illegal in a URL must be represented by its Unicode hexadecimal index preceded by the percent sign ("%"). Some of the illegal characters are as follows: space, <, >, single and double quotes, @, and &. For a complete list, see the URI specification at `www.ietf.org/rfc/rfc2616.txt`.

Overall Operation

The HTTP protocol is a request/response protocol. Most HTTP communications are initiated by a client, also known as a *user agent*. The user agent sends a request to a server (using a URL to identify the recipient of the request) and receives back a response. User agents are typically Web browsers, but they can also be HTML or XML editors, spiders, crawlers, and search engines.

In some cases, there may be intermediaries between the client and the server, as when the request goes first to a public proxy that rewrites parts of

the message and forwards it to the real server behind a firewall. Another common form of intermediary is a tunneling device that simply relays requests and responses without any changes. In this book, we actually use a tunnel (in Chapter 9) to watch the HTTP traffic between a SOAP client and a SOAP server.

The Structure of Client Request

An HTTP client request has three parts: the command line, the header section, and the entity body. For some commands, the body part is empty.

The HTTP request command line consists of a command (also called a *method*) followed by a URL and an HTTP version number, as in

```
GET /docs/info-page.html HTTP/1.0
```

This request uses the GET method to request the document specified by a relative URL, /docs/info-page.html. Absolute URLs can also be used, as in

```
GET http://www.ietf.org/rfc/rfc2396.txt HTTP/1.1
```

Immediately following the command line, the client sends headers with information about itself, including a list of document formats that it can accept. There is one header per line; each header consists of a name and a value separated by a colon, as in

```
User-Agent: Lynx/2.4 libwww/5.1k
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
```

The User-Agent keyword lets the server determine what browser is being used. This allows the server to send files that are optimized for the particular browser type. The Accept keyword will inform the server what types of data the client can handle. The data types are specified using the MIME protocol (*Multipurpose Internet Mail Extensions [MIME] Part Two: Media Types*, RFC 2046).

The command line and the subsequent header lines are all terminated by a blank line. If the method is POST, the blank line may be followed by additional data, which is usually intended for backend programs. Finally, another blank line terminates the request. A complete request might look like:

```
POST /axis/PrimeFactorsString.jws HTTP/1.0
Content-Length: 402
Host: localhost
Content-Type: text/xml; charset=utf-8
```

As you can see, the MIME type of XML documents is `text/xml`.

Two Ways of Sending Data to Backend Programs

Data can be sent to backend programs using either the GET or POST method. With GET, data is sent as the query string following the URL; with POST, it is sent in the body of the HTTP message. For small amounts of data (one or two arguments), the query string is a quick way of invoking a backend program without setting up an HTML page with a form (provided you don't mind the user seeing your data in the address window). For larger amounts of data, the POST method should be used.

Server Response

The HTTP response also contains three parts: the status line, the headers, and the body. The status line contains three fields: the HTTP version, a status code, and a status code description. The description is usually brief. For example, the status line

```
HTTP/1.0 200 OK
```

indicates that the server uses version 1.0 of the HTTP in its response, the status code is 200, and the meaning of the code is "everything's okay"; the request has been processed successfully. A few of the most common codes and their descriptions are listed below.

After the response line, the server sends headers with information about itself and the requested document. As in Request, there is one header per line, and each header is a colon-separated, name-value pair, for instance:

```
HTTP/1.1 200 OK
Date: Wed, 19 May 1999 18:20:56 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.7
Last-Modified: Mon, 17 May 1999 15:46:21 GMT
Content-Length: 10352
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

The Server keyword lets the browser know what server is being used. The date header will inform the client the time of the response (in terms of server time zone); the Last-Modified header will let the browser know the last time this document was modified, and, finally, the Content-type and Content-length will inform the browser of the properties of the document that is being sent. The server sends a blank line to end the headers.

If the client's request if successful, the requested data is sent. This data may be a static document or dynamically generated by a backend program. This result is called a *response entity*. If the client's request could not be fulfilled, additional data sent may be a human-readable explanation of why the server could not fulfill the request. The properties (type and length) of this data are sent in the headers. Finally, a blank line terminates the response. A complete response might look like the following:

```
HTTP/1.1 200 OK
<!-- headers as before -->
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<body>
<h1>Hello, HTTP</h1>
</body>
</html>
```

In HTTP/1.0, after the server has finished sending the response, it disconnects from the client and the transaction is over unless the client sends a `Connection:KeepAlive` header. In HTTP/1.1, the default is to keep the connection open so the client can make additional requests; to have it closed, the client sends an explicit `Connection:close` header. Because many HTML documents embed other documents as inline images, applets, frames, and so on, this persistent connection feature of HTTP/1.1 protocol saves the overhead of the client having to repeatedly connect to the same server just to retrieve a single page.

Request Commands (Methods)

Many methods are defined for HTTP 1.1, but only three of them—GET, POST, and HEAD—are widely used. We have already discussed GET and POST. The HEAD method requests only the headers to be sent, so the client can collect information about the server and the document named in the request without actually having it transferred. Otherwise, the headers of a HEAD request are the same as with GET, and the response is the same, except it has no body.

Less Commonly Used Methods and WebDAV

The remaining methods are less commonly implemented by current Web servers. They include:

- *PUT*: Requests that the server store the resource identified by the request URI.
- *DELETE*: Requests that the server delete the resource identified by the request URI.
- *OPTIONS*: Requests information about the communication options available on a server for the request URI, such as the methods that can be invoked on it. (Can it be deleted?)

These methods imply a vision of a read-write Web, in which the Web client is not only a viewer but also an editor of Web resources. For this vision to become reality, a host of difficult technical issues having to do with security and concurrency have to be resolved. These issues are addressed in WebDAV, a set of extensions to HTTP that is nearing completion at IETF. Some of the features supported by WebDAV are access control, version control, locking and unlocking of resources, namespace-awareness for copying and pasting XML data, and 128-bit encryption. Windows XP, Adobe Acrobat 5.0, and the Apache Web server provide support for WebDAV.

Server Response Codes

The HTTP server reply status line contains three fields: HTTP version, status code, and description in the following format. Status is given with a three-digit server response code. Status codes are grouped as shown in Table C-1.

Table C-1. Code Ranges and Their Meanings

CODE RANGE	MEANING
100–199	Informational
200–299	Client request successful
300–399	Client request redirected, further action necessary
400–499	Client request incomplete
500–599	Server errors

By far the three most common codes are as follows:

- *200 OK*: The request has succeeded. The server's response contains the requested data.
- *404 Not Found*: The server has not found anything that matches the request URI.
- *500 Internal Server Error*: The server encountered an unexpected condition, which prevented it from fulfilling the request.

The best status code is the one you don't see because the requested data is displayed.

APPENDIX D

Online Resources

WE GROUP ONLINE RESOURCES into several categories, as follows:

- standards
- sources of information, FAQs, and discussion lists
- sources of software
- everything else

We briefly mention the sources of the software used in the book, but see Appendix A for detailed information.

Standards

None of XML-related "standards" are literally so, in the sense of being approved by either an international standards organization such as ISO, or a national standards organization such as ANSI. They are all de facto standards produced by industry consortia. The most important of these consortia, by far, is the World Wide Web Consortium, W3C.

W3C Technical Reports

All W3C technical reports (recommendations at various stages, working drafts, and notes) can be found at `www.w3.org/TR/`. You can find any W3C technical document by searching through that page. Individual documents' URLs are formed by attaching an identifying string; for instance, ***Extensible Markup Language (XML) 1.0 (Second Edition)*** is found at `www.w3.org/TR/REC-xml`. Here are some of the URLs that are used in this book; we provide a title and the string to attach to `www.w3.org/TR/`:

- ***Extensible Markup Language (XML) 1.0 (Second Edition)***: `REC-xml`
- ***Namespaces in XML***: `REC-xml-names/`

- ***XSL Transformations (XSLT)***: `xslt`
- ***XML Path Language (XPath)***: `xpath`
- ***XML Linking Language (XLink)***: `xlink`
- ***XML Pointer Language (XPointer)***: `xptr`
- ***XML Schema Part 1: Structures***: `xmlschema-1`
- ***XML Schema Part 2: Datatypes***: `xmlschema-2`
- ***XML Information Set***: `xml-infoset`
- ***SOAP Version 1.2 Part 1: Messaging Framework***: `soap12-part1`

In addition to supplying technical reports, W3C is a source of WWW, HTML, and XML news, information, and public domain software. It also maintains a number of public mailing lists on its activities, found at `www.w3.org/Mail/Lists.html`.

OASIS Technical Committees

Another source of XML standards that is becoming increasingly important is OASIS (Organization for the Advancement of Structured Information Standards), `www.oasis-open.org`. The most important OASIS-sponsored project for this book is RELAX NG, but you might also be interested in its work on Docbook and ebXML, and their XML and XSLT conformance testing suites.

OASIS also provides a home for the Cover pages and `xml.org`, two important sources of information. (See the next section.)

Other Consortia

It seems that a business consortium has become a well-developed mechanism for creating de facto standards. We will mention a few here:

- UDDI.org, for developing UDDI, as described in Chapter 9.
- Web Services Interoperability Organization (WSIO), to ensure interoperability of Web services based on the SOAP, WSDL, and UDDI protocols. (See `www.ws-i.org/`.)

- WAPForum, for developing the Wireless Applications Protocol. (See `www.wapforum.org/`.)

- DAML.org, not an industry consortium but a consortium nevertheless, of a number of academic and research institutions, funded by DARPA, and committed to the vision of the Semantic Web.

Sources of Information

Our book investigates several broad topics: XML; Java XML processing; JSP, ASP, and Web applications; XML and databases; and Web services. The following resources are grouped according to these topics.

Large companies (such as IBM, Microsoft, Sun, and Oracle) have large stores of information on all of these topics. We particularly mention four:

- ***IBM Developerworks***, `http://www.ibm.com/developerworks/` has many tutorials, often with links to IBM alphaworks' site of free software. (IBM free software is not necessarily open-source.)

- ***Microsoft Developer Network (MSDN)***, `www.msdn.microsoft.com/`, and especially `www.msdn.microsoft.com/xml` and `www.msdn.microsoft.com/asp`, of course, has much useful information.

- Sun's Java site, `http://java.sun.com`, and especially `http://java.sun.com/xml`, `http://java.sun.com/jsp`, and `http://java.sun.com/j2ee`, are all good resources, with links to developer sites. There is also interesting software at `http://www.sun.com/xml/`.

- ***Oracle Technology Network***, `http://technet.oracle.com/`, has a lot of interesting information and software, including their recent "release candidate" of Oracle9i JDeveloper that integrates XML Java processing with relational database access. Be aware that the free download of this package is about 150MB, compressed.

XML Resources

The Cover pages (`http://xml.coverpages.org/`) are the oldest source of information on XML and, before it, SGML. They are maintained by Robin Cover (hence the name) and hosted by OASIS. It is a huge and diligently maintained resource; Robin Cover's summaries frequently provide useful insights.

While you are at or near OASIS, also visit `xml.org`, especially the FAQ page, `www.xml.org/xml/xmlfaq.shtml`. It has links to several XML and XSL FAQs, as well as FAQs for XML Schema, SOAP, and UDDI. It is also home to the xml-dev mailing list at `www.xml.org/xml/xmldev.shtml`. The list's archives at `http://lists.xml.org/archives/xml-dev` make excellent reading. More specific XML-related lists are at `www.w3.org/Mail/Lists.html`, and a very active XSLT mailing list is at `www.mulberrytech.com/xsl/xsl-list`.

Several XML-related resources are maintained by the O'Reilly network. The most important one is `www.xml.com`. It serves both as an online journal and a repository of information, with links to tutorials, reviews, and software.

For current XML events, visit `www.xmlhack.com` and perhaps subscribe to its daily newsletter. It will keep you up to date on everything XML, including the most passionate exchanges on the xml-dev list.

For Microsoft's coverage of XML, go to `www.msdn.microsoft.com/xml`. An abundance of information is available here, some of it Microsoft specific and some of general interest. More coverage of Microsoft-based XML processing can be found at `www.intsysr.com/` and `vbxml.com`.

Java XML Processing

There is a Java XML page at Sun, `http://java.sun.com/xml/`, with links to several specifications:

- ***Java Architecture for XML Binding*** (“***JAXB***”)
- ***Java API for XML Mes***saging (“***JAXM***”)
- ***Java API for XML Processing*** (“***JAXP***”)
- ***Java API for XML Registries*** (“***JAXR***”)
- ***Java API for XML-Based RPC*** (“***JAX-RPC***”)

Of these, ***JAXP*** is the only one in release status, and it's the only one that is used in this book. However, you should follow the development of the other specifications because they are likely to have a significant impact on the Web and Web services infrastructure.

JAXP APIs are distributed with most Java XML parsers, including Xerces. The Javadoc documentation for those APIs is part of both Xerces and Xalan distribution.

JAXP, of course, provides only a thin veneer of abstract classes on top of specific parsers. Parsers themselves produce objects that implement W3C DOM

interfaces to enable actual processing of XML data. In addition to Java implementations of DOM, there are other sets of Java-XML APIs that are more Java specific than language-independent DOM. Probably the best known of them is JDom (`www.jdom.org/`), which is currently in Beta 7. Designed specifically with Java in mind, these APIs are, presumably, easier to program and more efficient than JAXP APIs, but it is not obvious that they will prevail in the long run.

JSP, ASP, and Web Applications

For information on JSP, start from `http://java.sun.com/jsp`, where you will find multiple links to JSP-related information, both at Sun and elsewhere. For information on ASP, start from `www.msdn.microsoft.com/asp`.

In addition to Sun's JSP FAQ at `http://java.sun.com/products/jsp/faq.html`, there is an independent one at `www.esperanto.org.nz/jsp/jspfaq.jsp`.

Several ASP FAQs, including `www.aspfaq.com/`, collect information from Microsoft's ASP newsgroups.

XML and Databases

A starting point on XML and databases is, unquestionably, Ron Bourret's Web site, `www.rpbourret.com/xml/index.htm`. It has very valuable and up-to-date surveys on XML and databases, XML database products, XML data binding, and others. They serve as a comprehensive overview of the field.

An important, open-source XML database project, DBXML.org, has recently moved to Apache, becoming the Xindice project within `http://xml.apache.org`. Of all open-source native XML projects, this is probably the most important. Several commercial native XML projects are listed in Bourret's paper on XML database products. The Tamino product from Software AG is probably the most advanced, and it has to be significant that several Software AG employees are on critical W3C committees, such as XPath 2.0, DOM 3.0, and XQuery. The whole field of native XML databases will get a big boost when XQuery becomes a W3C recommendation, harmonized with XPath 2 and DOM 3.

Web Services

There isn't a single major source of XML ideas or software that is not involved with Web services in one way or another. Here is a brief listing:

- W3C is working on standards for SOAP and, more broadly, for XML protocols. W3C also has a note on WSDL and is actively working to position RDF as a Web services description language. There is a page at W3C on "Semantic Web web services resources," `www.w3.org/2001/11/11-semweb-webservices`.

- OASIS is working on XML messaging and a number of other projects within its ebXML initiative, joint with UN/CEFACT. (***UN/CEFACT*** stands for ***United Nations Centre for Trade Facilitation and Electronic Business***. Don't ask us how and why, but ***ebXML*** stands for ***Electronic Business using XML***. There is a dedicated Web site, `ebxml.org`.)

- Microsoft, IBM, Sun, Oracle, and Hewlett-Packard all work on various aspects of SOAP, WSDL, and UDDI. In addition to individual companies' sites, there are also UDDI.org (`http://uddi.org`)and the Web Services Interoperability Organization (`http://ws-i.org`).

Sources of Software

Our main source of software is the Apache foundation, `www.apache.org`. This is an umbrella organization for several open source projects, each with multiple subprojects. In particular, this book uses the following subprojects at `xml.apache.org`: Xerces (an XML parser that validates against DTDs and XML schemas), Xalan (an XSLT processor), and Axis (a framework for developing and deploying Web Services). This book also uses Tomcat and Log4J from `http://jakarta.apache.org`. Appendix A gives detailed instructions on how to download and install them. You should know that each of these projects has a mailing list for users and another one for developers. You can post questions to the users list and get answers, or you can answer other people's questions; eventually, you may want to join the developers list and contribute bug reports or code.

As we were writing the book, Microsoft's XML, ASP, and other Web-related technologies were, and still are, undergoing a massive change to the .NET framework, so the only Microsoft software we can recommend at this point is what is mentioned in Appendix A.

For software from IBM, Sun, and Oracle, go to the links listed in the beginning of the preceding section.

A small but interesting source of open-source software is W3C. In addition to Tidy (used and referenced in the book), they have a number of cutting-edge, proof-of-concept projects, all of them accessible from `www.w3.org/Status`.

No listing of XML standards or resources would be complete without a paragraph on James Clark's contributions. Before XML, he single-handedly created most of the open-source software for SGML (much of it used within commercial products as well). He was the technical lead of the committee that created XML, the editor of ***XSLT 1.0***, and co-editor of ***XPath 1.0***, simultaneously producing a reference implementation of the XSLT processor, xp. He went on to develop a grammar formalism for XML validation, TREX, that merged with Murata Makoto's RELAX and eventually became RELAX NG, an OASIS-sponsored specification that has been submitted to ISO. Visit the RELAX NG page at OASIS, James Clark's page for RELAX NG at `www.thaiopensource.com/`, and `www.jclark.com` for other software.

Other individual developers include David Megginson, `www.megginson.com`, the developer of SAX and SAX2, with multiple input from xml-dev; Rick Jelliffe, the inventor of Schematron, `www.ascc.net/xml/resource/schematron/schematron.html`; Dave Raggett, the author of Tidy; Michael Kay, the author of the XSLT processor Saxon (`http://saxon.sourceforge.net/`) and the editor of ***XSLT 2.0***; and many others that are too numerous to mention here. You will find them all on xml-dev.

Keep Looking

Any static page on Web resources, especially printed in a book, is going to be obsolete and incomplete soon. Keep looking through `google.com`, `altavista.com`, or `alltheweb.com` for other tutorials, code collections, or interesting developments. Visit `xml.com` and `w3c.org` once a week, subscribe to xmlhack newsletter, and peruse the archives of xml-dev and xsl-dev at least once a month. Things are moving fast, and they keep moving faster.

APPENDIX E

Troubleshooting in JSP

THINGS OFTEN GO WRONG. When they do, we get error messages. In a JSP, these messages are found in the console (that is, the Tomcat command line) window and in the browser, as an error page. In this appendix, we give a few suggestions on how best to use the information in the error messages, how to make your error pages more informative, and how to make your code more modular so that your error pages can better localize the problem.

> **NOTE** *If you are running Tomcat on a Windows machine, and you used the Windows executable installer, and you installed Tomcat as a service, then you won't have a console window. You can see error messages in the error page in the browser or in the log files in TOMCAT_HOME/logs.*

This appendix consists of four sections. In the first, we will look at the error messages in the console window and at the Java code that generates them. As we explained in Appendix B, JSPs get rewritten as servlets and compiled. Both the source code of the servlet and the compiled code are placed in a subdirectory of the TOMCAT_HOME/work/localhost directory.

In the second section of this appendix, we look at error pages; in the third, we explain how to make JSPs more modular. In the fourth and last section, we consider the biggest source of frustration for JSP and all Java programmers: classpath problems.

Looking at Servlet Code for JSP

In JSP, error messages frequently show the line numbers in the JSP or in the generated servlet. Let's start with a bad JSP file, webapps/xmlp/bad/badCode1.jsp, as shown in Listing E-1.

Listing E-1. Bad JSP 1

```
<html><head><title>Blowups Happen # 1</title></head><body>
Time to explode <% bad code %>
</body></html>
```

What do we get from `http://localhost:8080/xmlp/bad/badCode1.jsp`? An error message, of course, specifically from `org.apache.jasper.compiler.Compiler.compile(Compiler.java:284)`; it says

```
org.apache.jasper.JasperException: Unable to compile class for JSP
An error occurred at line: 2 in the jsp file: /bad/badCode1.jsp
Generated servlet error:
C:\tomcat401\work\localhost\xmlp\bad\badCode1$jsp.java:58: ';' expected.
                  bad code
```

Notice that two files are involved: we are at the second line of /bad/badCode1.jsp, but the 58th line of a generated Java file, whose filepath is given in full: it's a subdirectory of TOMCAT_HOME/work. All servlets generated from JSPs end up in subdirectories of the TOMCAT_HOME/work directory.

Looking at that file, we can find the generated code defining a class named `badCode1$jsp`. The code starts by saying that this class lives within a package (and the compiler should give it access to other classes in that package); that all the classes within the existing packages `javax.servlet`, `javax.servlet.http`, and others will be imported; it then declares the `badCode1$jsp` class as an extension to a base class named `HttpJspBase`, and starts declaring methods within `badCode1$jsp`, as shown in Listing E-2.

Listing E-2. Generated Servlet for Bad JSP 1

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;
public class badCode1$jsp extends HttpJspBase {
...
public void _jspService(HttpServletRequest request, HttpServletResponse  response)
 ...
```

To make sense of this, look at the Tomcat documentation, or your JDK documentation, or simply look at `www.google.com/search?q=HttpJspBase`, where you'll see API documentation for the class that's being generated for you. Of course, that documentation will not cover the code that's extracted from the JSP page "badCode1.jsp" itself:

```
out.write("<html>......<body>\r\nTime to explode ");
bad code
out.write("\r\n</body></html>\r\n");
```

Indeed, that's bad code. If we supply the ";" that the error message claimed to be expecting, replacing `bad code` with `bad code;` inside the JSP file and then clicking on Refresh/Reload in the browser, it will still be bad code. Now the error message will be

```
Class org.apache.jsp.bad not found.
```

The statement `bad code;` could be a legal declaration of a variable named `code` which is of class `bad`, just as `String str;` declares a variable of class `String`, but there is no class named `bad` within the JSP package we're putting our page into. If we defined such a class, we could make this run.

Error Pages

We'll start with the bad/error.jsp page we suggested in Appendix B, Listing B-7. (See Listing E-3.)

Listing E-3. Simple Error Page (Same as Listing B-7)

```
<%@ page isErrorPage="true" %>
<html><head><title>ErrorPage</title></head><body>
Problem: <%= exception.getMessage() %>
</body></html>
```

This page will be activated by any uncaught error in any JSP for which it is the error page. We are going to make an error page for bad/badCode2.jsp, shown in Listing E-4. It expects a parameter called `count`, compiles fine, and generates a one-row table counting from 0 to count-1.

Listing E-4. Counting JSP

```
<%@ page errorPage="error.jsp" %>
<html><head><title>Blowups Happen #\ 2</title></head><body>
<table border="1"><tr>
<%
  int count=Integer.parseInt(request.getParameter("count"));
 for(int i=0;i<count;i++) { %>
    <td> <%= i %> </td>
 <% } %>
</tr></table>
</body></html>
```

If we connect to `http://localhost:8080/bad/badCode2.jsp?count=5`, we will get a row of numbers from 0 to 4. Let's see what will happen if we take off `?count=5`, providing no parameter:

```
<html><head><title>ErrorPage</title></head><body>
Problem: null
</body></html>
```

This error message is a bit too cryptic. We can do better if we expand the error page, as in Listing E-5.

Listing E-5. A Better Error Page

```
<%@ page isErrorPage="true"
    import="java.io.PrintWriter" %>
<html><head><title>ErrorPage</title></head><body>
Problem: <%= exception.getMessage() %>
<%
  exception.printStackTrace();  // send details to console
%>
<br/>Detail:<br/>
<textarea rows="15" cols="80">
<%
  exception.printStackTrace(new PrintWriter(out,true)); // to page
%>
</textarea>
</body></html>
```

Now the error page says `exception.printStackTrace();` and that sends a detailed report to the console (the Tomcat command line window); it also sends the same report to the normal output page, within a text area. We can read it either way; the report begins with

```
java.lang.NumberFormatException: null
          at java.lang.Integer.parseInt(Unknown Source)
          at java.lang.Integer.parseInt(Unknown Source)
          at org.apache.jsp.badCode2$jsp._jspService(badCode2$jsp.java:59)
```

This will generally be enough to get us started debugging. We can extract more information from the `Exception` object, but it may be a better strategy to do more to prevent errors, instead of having to discover them in the error page.

Writing Modular JSPs

Errors are usually easiest to handle if you localize them as much as possible. One way to localize possible errors is to break up a monolithic JSP into smaller modules. In particular, we can isolate the code for counting into a Java method definition outside the HTML template, and have the JSP code within the HTML template call that method. Consider goodCode.jsp in Listing E-6.

Listing E-6. JSP with a Method Definition

```
<%@ page errorPage="error.jsp" %>
<html><head><title>Modular JSP</title></head><body>
<table border="1"><tr>
<%= countUp(request.getParameter("count")) %>
</tr></table>
</body></html><%!
public static String countUp(String countStr){
 try{
  int count=Integer.parseInt(countStr);
  StringBuffer sB=new StringBuffer();
  for(int i=0;i<count;i++)
    sB.append("<td>").append(i).append("</td>");
  return sB.toString();
 }catch(Exception ex){
   return "<td>countUp('"+countStr+"'): "+ex+"</td>";
 }
}
%>
```

The first part of this JSP, down to </body></html>, is much simplified and obviously correct. It refers to a countUp method that produces the string “<td>0</td> . . . <td>12</td>" for count=13. That method is no longer strictly JSP; it cannot use HTML inclusions the way that the same code could before, and this is a loss. However, the loss is offset by a gain: exceptions, if any, will arise within the simple context of a short method definition. Consider what happens if we call goodCode.jsp with count=abc: it produces a nice one-row, one-cell table containing this message:

```
countUp('abc'): java.lang.NumberFormatException: abc
```

That may not be what you want your end user to see, but it does tell you just what’s going on. Much of our JSP code in this book is divided into method and class definitions that are used within the JSP code proper, as part of the HTML template.

Classpath Problems in Java/JSP

A large fraction of the problems you will encounter in running or recompiling our code are "classpath problems": you are invoking a library in a JAR file, and `java` or `javac` can't find it. Worse, Java seems to find it but is in fact finding the wrong version of the library; for example, you may have several copies of xerces.jar. The situation is better than DLL hell, and we have simplified it further by putting all our JARs into TOMCAT_HOME/common/lib, but you should be aware of it. If you see a "package not found on import" error, or a "cannot resolve symbol" error, it's quite likely that a JAR is in the wrong place or that you have not restarted Tomcat since putting it in the right place.

To find which JAR or JARs have definitions for a given class, it helps to understand that a JAR file is a zip archive, a compiled Java class or interface is a `.class` file, and filenames are stored in uncompressed format. To find which jars have definitions of the Node interface, you can search (with `grep` on Linux or `Ctrl+F` on Windows) for JAR files containing the string `Node.class` or even, if you know specifically that you are looking for `org.w3c.com.Node`, for the string `org/w3c/dom/Node.class`. You probably have a fair number of these, mostly copies of xerces.jar or xalan.jar. Look at dates and sizes to see which one you want (usually the latest) and put it ahead of the rest of them on the classpath and perhaps remove the rest from your system.

Usually, especially with our usage of common/lib (all required JARs are in one flat directory), that's enough. Sometimes it's not, because *classpath* is really a misnomer: the term *classpath* actually refers to a hierarchy of class loaders that invoke each other, and the top class loader (for our purposes, the one that loads all JARs in your JDK's jre/lib/ext directory) is visible to all others, but not necessarily vice versa. Here is an example of how this may become relevant.

Suppose that you developed some Java classes for your Web application and placed them into your webapp's WEB-INF/lib subdirectory. This is good for distributing your Web application as a single self-contained archive that can simply be expanded into the TOMCAT_HOME/webapps directory. However, you should be aware of the possibility that Tomcat's common libraries in common/lib are visible to your webapp's libraries, but not vice versa. This is a good design principle: each class loader effectively defines an extension to the Java language. Usually it will cause no problems: for example, your own WEB-INF/lib directory can contain classes that refer to common/lib/xalan.jar. However, if you have written XSLT extension functions that are called by Xalan, then you want your extension functions to be visible to Xalan's class loader. At that point, you have to start thinking about hierarchies of class loaders, and you may well end up using more than one copy of xalan.jar.

Remember to check the readme.htm file of our code archive.

Index

Numbers and Symbols

19 Short Questions about Namespaces (with Answers) (David Megginson), 75
@* abbreviation, in path expressions, 220
@ character, referencing XPath attributes with, 25
& (ampersand) character, predeclared character reference for, 48–49
‘ (apostrophe) character, predeclared character reference for, 49
* (asterisk) character, in EBNF rules, 50
/ (backslash) character,
  for referencing the root of the document tree, 25
  using in path expressions, 215
: (colon) character, use of in XML, 55
, (comma), as sequence separator in children-only content model, 89
© (copyright symbol), entering into an XML document, 48
{} (curly brackets), denoting an XSLT variable to be evaluated with, 29
> (greater than) character
  predeclared character reference for, 49
  in XML end tags, 4
  in XML start tags, 4
< (less than) character
  predeclared character reference for, 49
  in XML start tags, 4
* node test, 215–216
% (percentage) symbol, contained in PE declaration and reference, 100
. (period) abbreviation, in path expressions, 221
| (pipe) operator
  as choice separator in children-only content model, 89
  using to combine createNodeIterator constants, 184
  union expressions formed with, 218–219
+ (plus) character, in EBNF rules, 50
? (question mark) character, in EBNF rules, 50
" (straight quotes)
  predeclared character reference for, 49
  using to specify a value as a literal string, 234
* test, and the namespace, 216–217
.. abbreviation, in path expressions, 221
.//d abbreviation, in path expressions, 221
// abbreviation, in path expressions, 220
:: separator, using in path expressions, 215
<=, <, >=, > operators, in XPath, 208
</ characters, use of in XML end tags, 4
=, != operators, in XPath, 208

A

abbreviated axis specifiers, of path expressions, 219
abbreviated paths, of path expressions, 220
abbreviated predicates, of path expressions, 219
abbreviated steps, of path expressions, 219–220
abstract modules, and implementation, 110–111
acceptNode() method, 184
accumulation list processing pattern, 280
action files
  for the application, 321–322
  for refset operations, 339
  for update queries, 339
ActiveSAXbookpicker.asp
  application’s code, 188–192
  code example, 188
  ContentHandlerImpl class in, 189–190
  SAX callbacks in, 190–193
  startDocument() code example, 191–193
actor attribute, in Envelope namespace, 460–461
add action, refset, 353
addCitation() method
  for adding a citation and submission record to a database, 350–353
  definition code example, 351–353

additional predicates, 217
addRefs() method, refset, 353
addTreeAddr()
  recursive from addTreeAddrRec.jsp, 180
  stack-based from addTreeAddrCore.jsp, 179
all-reachable query
  code example of root and closure templates, 251–252
  standard procedure for, 249–250
ampersand (&) character, predeclared character reference for, 48–49
ancestor axis, XPath, 212
ancestor nodes, of tree, 3
ancestor-or-self axis, XPath, 212
and operator, in XPath, 208
ann elements, content model of, 317
annotation elements, 310–311
  use of XLink structures with, 316–318
annotation vocabulary, for DTD compatibility, 380–381
anonymous types, example defined within the element definition, 396
ANY content model, for DTDs, 88–89
Apache Axis, using as framework for Web services, 425–490
Apache Tomcat, Web server and JSP processor as part of XSLT setups, 21
Apache Xalan, XML processor as part of XSLT setups, 21
Apache Xerces, XML parser as part of XSLT setups, 21
appCache, function of, 326
application
  components and files, 320–322
  search actions supported by, 320–321
  the structure of, 320–331
  typical scenario for user interactions, 322
  update actions supported by, 321
application cache, instantiating, 325–326
application (re)loading, and session initialization, 326–328
apply-imports instruction element, XSLT processing model, 225
apply-templates instruction element, XSLT processing model, 225
arc elements, descriptive content for, 317
arc value, for xlink:type for children elements, 62
arcs, in directed graphs, 60
arcs template, for outputting arcs in XLink application, 243
array accessors (numerical indices), referenced in SOAP1.2-2 Section 4, 465–466
ASP
  JDBC equivalents in, 337–338
  in Web applications, 503–505
ASP-based Web application, code example for validation, 85–86
ASP code
  to run XSLT, 23–24
  for xptrans.jsp, 204–205
ASP page, code example for ActiveSAXbookpicker.asp, 188
asterisk (*) character, meaning of in EBNF rules, 50
attribute axis, XPath, 212
attribute declarations, EBNF format, 90–91
attribute defaults, 92
attribute instruction element, XSLT processing model, 225
attribute nodes, XPath, 210
attribute-set element, XSLT processing model, 225
attribute values, 91
attributeGroup element, XML Schema, 420
attributes
  common way of processing, 136–137
  and namespaces, 59–60
  outputting with computed values, 28–29
  patterns for declaring, 421
  redefining in a DTD, 105–106
  in XPath, 25
Attributes interface, SAX2, 136–137
authorization token request, code example, 485–486
authorization token response, code example, 486
Axis. *See* Apache Axis
Axis User Guide
  finding details of TCPMon operation in, 445
  Web site address, 471

B

backend processing, improvements to, 503
backend programs, sending data to, 512
basic-form.rng vs. form.rng, 388–390
Basic module, form element defined in, 388
beanMapping elements, function of, 471

behavioral markup vs. semantic markup, 63–64
Berners-Lee, Tim, HTML defined by, 13
Bible, XLink code example, 65–69
binary split. *See* tree recursion (binary split)
binary split vs. linear recursion, summary of test results, 292
Block.class
- defined in the Text module, 389
- redefined in the Form module, 389–390

Block.mix, defined in the Text module, 388
Block.model, defined in the Text module, 388
body element, referenced by the noframes element, 387–388
Book Picker, as DOM builder, 186–193
BookPicker class
- code example, 155–156
- code for output, 156–157
- vs. VersePicker class, 154

boolean() conversion function, in XPath, 206
Boolean data type
- possible values for, 206
- supported by Jing processor, 393

Boolean operators, in XPath, 208
Bourret, Ronald
- *Namespace Myths Exploded* by, 75
- *Namespaces FAQ* by, 75

Bray, Tim, *XML Namespaces by Example* by, 75
browsers
- approaches to displaying XML in, 35
- code example of HTML document in, 43
- implementation of XML parsing and CSS styling by, 35
- XSLT support on the client side by, 37–39

business entity request and response, code example, 487–489

C

C struct, using for office equipment inventory example, 463–464
call-template(with-param) instruction element, XSLT processing model, 225
callback. *See also* event handler
- defined, 128
- illustration, 129

candidate recommendations (CRs), by W3C, 13–14
canonical representation, 394
Cascading Stylesheets (CSS). *See also* CSS (Cascading Stylesheets)
- development of, 30–31

CDATA (Character DATA) attribute type, 91
CDATA sections, entering into a document, 49
character entities, representing by their Unicode numbers, 48–49
characters() callback, for verse picker application, 146
child axis, XPath, 212
child nodes, of tree, 3
children-only content model, code examples of, 89–90
choice separator, pipe character as in children-only content model, 89
choose(when) instruction element, XSLT processing model, 226
CIA World Factbook. *See* The World Factbook
citation elements
- progression of steps in creating and modifying, 340
- storing in a database, 319
- in XML data, 315

citation input form
- outline of XSLT parts, 344
- XSLT for, 342–347

citations, 310–311
- code example, 342–343
- code example for computing ID, 348–349
- creating and editing, 347–353
- example of entry form for, 343

Clark, James
- contents of his RELAX NG Web site, 363
- creation of DSSSL stylesheet language by, 33–34
- Web site address for a separate frames DTD discussion, 109
- *XML Namespaces* by, 75

.class PE name, to represent elements of similar meaning, 112
clear action, refset, 353
client-side validator, error message from, 83
client variations
- code example of another kind of client, 449
- SOAP as XML, 448
- in Web services programming, 448–456
- close-by axes, XPath tree, 211

Codes for the Representation of Names of Languages (ISO 639), 59
colon (:) character
  use of in XML 1.0, 55
  use of in XML Namespaces, 55
comma (,) character, as sequence separator in children-only content model, 89
command line, using to validate RELAX NG grammars, 368–369
comment instruction element, XSLT processing model, 225
comment nodes, XPath, 210
committed record, code example, 319
committers, defined, 318
Common.attrib, defined, 112–113
compiler-compiler, defined, 363
complex types
  code example for declaring, 422
  patterns for defining, 421
computed values, outputting attributes with, 28–29
conditional processing, 27–28
conditional sections, 106–107
conflict resolution, and default templates, 232–233
connection pooling, 333–334
connections and connection pooling, 332–334
constraining facets
  complete list of, 398
  for data types, 394–395
  for deriving data types, 395
content handler
  as part of SAX trio, 128
  for SAX filters, 144–145
content handler definition, code example, 140
content model, of ann, 317
content model file, XHTML Basic DTD, 117–119
content models, types of, 7
ContentHandler interface, code example of the callback declarations, 135
ContentHandlerImpl class
  in ActiveSAXbookpicker.asp, 189–190
  code example of enumeration, variables, and properties, 190
context of evaluation, and the XPath data types, 199–201
.content PE name, to represent the content model of an element type, 112
copy(shallow copy) instruction element, XSLT processing model, 226
copy-of (deep copy) instruction element, XSLT processing model, 226
copyright symbol (©), entering into an XML document, 48
Core interfaces
  add-address page using (stack or recursive), 181
  in DOM Level 1 and 2, 171
Costello, Roger
  Web site address for Schematron information, 418
  XML Schema materials developed by, 401
count(node-set), node-set number-valued function, 221
CountTags
  code procedure, 138
  entry page, 137
cp symbol, defined, 89
createNodeIterator() method
  code example, 183
  filtering parameters of, 184
creation methods arguments, 183
CRs (candidate recommendations), by W3C, 13–14
CSS (Cascading Stylesheets)
  example of simple, 32
  limitations of, 33
  for XML, 31–32
curly brackets ({}), using to denote an XSLT variable to be evaluated, 29

D

data formats, describing with XML, 1
data linking, describing with XML, 1
data processing, describing with XML, 1
data sets, generating large in XSLT, 284–292
data transfer, describing with XML, 1
data transformations, describing with XML, 1
data types
  built-in in XML Schema Part 2, 392–393
  describing with XML, 1
database structure, of relational for storing XML data, 319–320
datatypeLibrary
  Jing processor support for, 393
  and XML Schema data types, 392–400
DCMES DTD, 314–315
descendant axis, XPath, 212

decimal-format element, XSLT processing model, 225
declarations. *See* namespace declarations
DeclHandler, exposure of individual declarations in the DTD by, 130
decorative axes, XPath tree, 211
decorative nodes. *See* attribute nodes; namespace nodes
default templates
  and conflict resolution, 232–233
  and node types, 232
definitions
  combining multiple in RELAX NG, 383–384
  replacing in RELAX NG, 384
delete action, refset, 353
delRef() method, refset, 353
DeRose, Steve, XPath development by with James Clark, 34
Dertouzos, Michael, joint creator of the W3C, 13
descendant axes, XPath tree, 211
descendant nodes, of tree, 3
descendant-or-self axis, XPath, 212
descriptive content, for locator and arc elements, 317
directed graph, arcs in, 60
display action, refset, 353
distinct locators, XSLT, 260–262
divide-and-conquer problem solving, 286–287
divisions, code example for keys, 273
DLL, installing, 497
document, and its DTD, 93
document() function
  using within an XSLT stylesheet, 296–297
  in XPath, 69–70
document instances, schema-related markup in, 423–424
Document interface
  DOM (Document Object Model), 170
  IDL code example, 175
  Java code example, 176
document-matching schema
  with elementFormDefault=qualified, 415–416
  with elementFormDefault=unqualified, 416–417
document-oriented XML vs. data-oriented XML, 19–20
document tree
  code example of recursion down, 298
  tree recursion based on, 297–299
Document Type Definition (DTD). *See* DTD (Document Type Definition)
DocumentBuilder (DOM parser), JAXP class, 133
DocumentBuilderFactory class, JAXP, 133–134
DocumentFragment interface, 173
documents
  reuse of DTDs with an internal subset in, 102–105
  validating namespaced, 103–105
DocumentType interface, 173
DOM (Document Object Model)
  a brief history, 170–171
  defined, 6
  main interfaces, 170
  other interfaces and classes, 174–175
  treatment of XML documents by, 125–126
DOM APIs, functions included in, 6
DOM builder, Book Picker as, 186–193
DOM Core interfaces. *See* Core interfaces
DOM interfaces, overview, 172–176
DOM Level 1, parts of, 171
DOM Level 2, interfaces specified by, 171
DOM Level 3 module, Web site address for working draft, 173
DOM Node types, table of, 173
DOM programming, 169–181
DOM tree, code example for building, 349–350
DOM tree address, simple example, 176–181
DOMBuilder class, code example for ParseURL() function, 189
DOMBuilder object, code example for creating, 187–188
DOMException, 174
DomImplementation interface, testing for optional interfaces with, 171
DOMString, 174
DOMtreeAddr/treeAddr.jsp application
  code of main page, 178–179
  outline, 177–178
double data type
  constraining facets, 395
  example, 394
dreams.xsl, code example, 68
driver file, XHTML Basic DTD, 114–117
drivers, installing and using JDBC, 331–332
DSSSL stylesheet language, created by James Clark, 33

DTD (Document Type Definition)
- attributes and, 17–19
- declaration of entity names in, 93–94
- a document with an external, 18
- everything contained in, 120
- with external parsed entities, 96–97
- facilitating transition from RELAX NG to, 380–381
- and namespaces code example, 104
- problems with, 120
- and RELAX NG correspondences, 370
- vs. RELAX NG grammar, 364
- reuse with an internal subset in a document, 102–105
- reuse within another DTD, 105–106
- syntax and code examples, 88–93
- used for writing grammar rules, 3
- and validation, 81–124
- XHTML Basic, 114–119
- XML code examples without, 15–17

DTD compatibility package, features provided by, 381
DTD-less documents
- code example of all you can find in, 51–52
- kinds of material contained in, 49–50

DTDHandler, 130
Dublin Core, 313–315
Dublin Core Metadata Element Set (DCMES). *See* Dublin Core
Dublin Core RFD/XML, code example, 314

E

EBNF (Extended Backup-Naur Form), use of, 50–51
EBNF rules/productions
- operators used in, 89
- and the syntax of names, 55

echoContentHandler, 150–151
ECMAScript, code example of the Document interface in, 176
element c, code example of type definition of the type of, 405
element content, types of, 7
element declarations and content models, for DTDs, 88–90
element instruction element, XSLT processing model, 225
element nodes, XPath, 210
element tree, of an HTML document, 43
elementFormDefault attribute, using, 415–417
elements
- vs. attributes and attribute modifications, 105–106
- code example for declaring, 420
- code example of redefining, 105–106
- interleaving of mixed-content model and, 375–376
- patterns for declaring, 420

elements and attributes
- defining as lists in RELAX NG, 375
- expressing a choice between, 374
- uniform treatment of in RELAX NG, 373–376

embed use case, XHTML modularization, 110
EMPTY content model, for DTDs, 89
Encoding Dublin Core Metadata in HTML (RFC 2731), code example from, 313
encodingStyle attribute, in Envelope namespace, 460–461
end tag, format of, 4–5
endDocument() method, use of in ContentHandler interface, 135
endElement() method, use of in ContentHandler interface, 135–136
extended link element, possible values for children elements of, 62
extension functions
- vs. external Web applications, 297
- XSLT/XPath, 71–73

entities
- defined, 93
- general and parameter, 93–101

entity, defined, 48
entity declarations, defined, 93
entity references, defined, 93
entity resolution, 95
EntityResolver, function of, 130–131
env:Body element, 459, 461-462
Envelope namespace, 459
- attributes in, 460–461

env:Envelope element, 459
env:Fault element, mandatory and optional children in, 461–462
env:Header element, in the abstract Message Exchange Model, 459–461
error handler, code example for obtaining, 367
error pages, troubleshooting in JSP, 527–528
ErrorHandler, function of, 130
example, English-language for syntax diagrams, 3–4
exclude-result-prefixes attribute, using, 267

executeUpdate(), using for UPDATE queries, 337
extend use case, XHTML modularization, 110
extended link element
  code example of XML structure of, 63
  and its XLink graph, 62–64
extension elements and functions, longest verse using, 299–303
extension functions
  vs. external (Web application) functions, 296–297
  time of range generation by, 295–296
  using to generate a range, 293–296
  using within XSLT, 293–297
  when they are useful, 293
external DTD references, code example, 93
external entity references (including DTD references), 97–99
external (Web application) functions vs. extension functions, 296–297
external general parsed entities, 93
  DTD with, 96–97
  uses for, 99
external Web applications vs. extension functions, 297
externalRef pattern, processing in RELAX NG, 383

F
facets, for data types, 394–395
factors() method, server stub code example for, 444–445
factorSoapEnv() function, 454
fallback instruction element, XSLT processing model, 226
Fault elements, SOAP, 452
filter writing, SAX support for, 144–145
filtering, a list by the prime() predicate, 280
find-neighbors query
  root template, 246–247
  XLink application, 246–249
#FIXED keyword, using with attribute defaults, 92
fixed-width records, converting to XML, 163–169
FixedWidth.jsp program
  code example with gaps, 165
  outline, 164
FixedWidthReader class
  outline, 166
  variables, constructor, methods other than parse(), 167
flat structures, converting to hierarchical, 273–279
following axis, XPath, 212
following-sibling axis
  for building a hierarchy recursively, 278–279
  XPath, 212
for-each instruction element, XSLT processing model, 225
For loop, using to process attributes, 136–137
form element, defined in the Basic module, 388
Form module, Block.class redefined in, 389
formal (or uninterpreted) languages, XML languages as, 2
Formal Public Identifiers (FPIs), in SGML, 98
form.rng vs. basic-form.rng, 388–390
Forms and Tables, Basic and non-Basic modules for, 388–392
frames, problem in XHTML modularization, 386–387
frames page, for application, 323–325
freeConnection(), code example of rewritten to use connection pooling, 333
Function Call, XPath, 207
functions and function libraries, 207–208
fundamental facets, for data types, 394–395

G
general entity names, five predeclared, 93
general entity references, use of, 93
general parsed entities, 94
general unparsed entities, 94
generate-id() function, 261
getChildren() function, in DOM APIs, 6
getConnection() from getConn.jsp, code example for, 333
getConnection() function, code example for obtaining a connection, 333
getConnection() method, of DriverManager class, 332–334
getDateFormat() function, code example, 329–330
getDBF(), code example, 327
getFactors() function, writing, 450–451
getParent() function, in DOM APIs, 6
getRefIf() method, XPath filtering by, 354

getTransformer() function, code example, 329–330
getWebAppPath() function, code example, 329–330
global attributes, providing specific functionality with, 59
global declarations vs. local declarations and element references, 413–414
globally unique names, creating, 53–55
grammar, defined, 3
grammar rules
  standard notations used for writing, 3
  of XML languages, 6–9
graphs. *See* XLink graphs
greater than (>) character, predeclared character reference for, 49
greeting element, of Hello, XML, 4
group element, XML Schema, 420
group/XSLT/DivReg.xsl, code for processing data rows, 267–268
grouping and tables, 263–272
grouping problem, discovery of by Steve Muench of Oracle, 262–263
groupXSLT/DivReg.xls data, converting to HTML, 265–266

H

hashtable, instantiating to hold the refset, 325–326
hashtable data, code example for converting to XML data, 158–159
hashtable parser, for XML output, 158–159
hashtable parser code, defining a class to implement XMLReader interface, 159–163
HashtableParser, the real parse() method for, 160–161
HashtableReader
  code example of variables and a constructor for, 159
  converting from SAX to text with markup, 161–163
Hello, XML
  simple syntactical code example for, 4–5
  syntactical diagram of, 5
hierarchical structures, converting flat to, 273–279
hierarchies
  building recursively, 278–279
  templates for building using xsl:key and generate-id(), 277–278
  using xsl:key and generate-id() to build, 276–277
HTML
  compared to XML and XHTML, 41–48
  defined by Tim Berners-Lee, 13
  evolution of, 109
  as a language, 44–45
  vs. XHTML, 42–44
  vs. XML, 45
HTML document
  in a browser, 43
  with a different stylesheet, 45
  element tree of, 43
  program for converting to XHTML, 47
HTML file, code example for client-side validation, 84–85
HTML file structure, of *The World Factbook*, 275–276
HTML form, code example, 328
HTML form with GET method, code example of request corresponding to, 501
HTML form with POST method, code example of request corresponding to, 502
HTML hyperlink
  similarities to an XLink simple link element, 62
  vs. an XLink simple link element, 61–62
HTML page, that submits data to PassAlong.jsp, 453–454
HTML-to-XHTML conversion program, by Dave Raggett, 265
HTTP protocol, 509–515
  less commonly used methods and WebDAV, 514
  overall operation of, 510–513
  request commands (methods), 513–514
  server response, 512–513
  server response codes, 514–515
  the structure of client request, 511–512
HTTP tunneling, 445
hypertext links, defined, 61

I

IANA (Internet Assigned Numbers Authority), language identifiers registered with, 59
IBM's Web Services ToolKit (WSTK), Web site address for downloading, 431, 480
ID attribute value, 91
id() function, implementing make-row with, 240

id() node-set function, 221–222
IDL code example, of the Document interface, 175
IDREF attribute value, 91
IETF (Internet Engineering Task Force), 13
if instruction element, XSLT processing model, 226
IGNORE keyword, for conditional sections, 106–107
#IMPLIED (optional) attribute default, 92
import and include elements, XSLT processing model, 225
import element, XML Schema, 419
import libraries, for verse picker application, 147
imports and beans, login check, sources of, 325
include element, XML Schema, 419
INCLUDE keyword, for conditional sections, 106–107
include pattern, processing in RELAX NG, 383
Included files
  for application, 321–322
  defined, 339
Infoset, overview, 121–123
infoset augmentations, 92–93
infoset specification, for XML standards, 6
inline elements section, of content model file, 117–118
input source
  converting the file name to a URL, 132
  giving to the parser, 131–132
  as part of SAX trio, 128
InputSource, and other sources of input, 131–132
InputSource object
  creating, 131–132
  returned by resolveEntity() method, 130–131
INSERT statement, code example of for a citation, 351
installation guide, 491–500
instruction elements
  summarized for XSLT processing model, 225–226
  for XSLT children of xsl:template, 223
integer array, code example for converting string of tokens to, 166
integer division, code example, 271
integer types, supported by Jing processor, 393
Interface Definition Language (IDL), in DOM, 172
internal general parsed entities, uses for, 99
Internet Explorer (IE), implementation of XML parsing and CSS styling by, 35
Internet Explorer and MSXML support for XSLT, 38
ISO 639. *See Codes for the Representation of Names of Languages* (ISO 639)
ISO 8879:1986, international standard for SGML, 13
isSupported(feature,version) method, of Node interface, 174

J

Java
  code example of the Document interface in, 176
  range function, 294–296
Java class
  calling a static method of, 71
  converting to a SOAP server, 434
Java code
  compiling, putting in a JAR, and copying where Tomcat can find it, 469–470
  for Office Equipment Web service, 468–470
Java extension functions, 303–307
Java framework, installation guide, 492–495
Java implementation
  of get properties and delete business, 483–484
  of main() method, 481–482
  of proxy, token, and new business, 482–483
Java/JSP, troubleshooting classpath problems in, 530
Java methods, definitions of, 329–331
Java servlets and JSPs, in Web applications, 505–507
Java system functions, for time and memory, 292
Java validator, running, 368–369
Java XML processing, online resources for, 520–521
Java-XML tutorial, Web site address, 150
java.net.URL constructor, server stub code example for, 443
JavaScript extension function, code example in Xalan, 302–303

JAXP
  code for creating a parser, 134
  creating a parser without using, 142
  parsing with, 140–141
JAXP classes, for creating parsers, 133–134
JAXP Transformer object, XSLT transformations performed by, 152
JDBC (Java Database Connectivity), 331–338
  equivalents in ASP, 337–338
  installing and using drivers, 331–332
  PreparedStatement as alternative to using Statement, 336–337
Jing data types page, clarifications and caveats from, 393
Jing validation tool
  using to obtain an XML Reader, 366
  validating derived types with, 400–401
  versions, 365
JScript code, for invoking a SOAP action, 449–450
JSP
  looking at servlet code for, 525–527
  and the socket connection, 454–456
  troubleshooting error pages, 527–528
  troubleshooting in, 525–530
  writing a modular, 529
JSP, ASP, and Web applications, online resources for, 521
JSP code
  for outputting hashtable contents using Java and JSP, 141
  to run XSLT, 24
  for SOAP client, 439
JSP page, with XPath, 73–74

K

key element, XSLT processing model, 225
key() function, and the xsl:key element, 259–260
keys, building a hierarchy with, 276–278
King James Bible, using as XML data for an XLink application, 196–198

L

language binding, provided for Java and JavaScript (ECMAScript), 172
language codes, defined in *Codes for the Representation of Names of Languages,* 59
language type, supported by Jing processor, 393
last(), node-set number-valued function, 221
leaf nodes, of tree, 3
less than (<) character, predeclared character reference for, 49
lexical space, 394
LexicalHandler, access to lexical material provided by, 130
linear recursion
  code example of range by, 286
  defined, 283
  vs. tail recursion, 284
  vs. tree recursion (binary split), 285–292
link element, defined, 60
Link Source document, code example, 197
link source to linkbase transformer, tasks it has to perform, 242–246
Linkbase, from Link Source to, 196–198
Linkbase document, code example, 197–198
linkTransform.xsl, code of, 68–74
list-helpers template
  improving efficiency of, 238
  as a wrapper for make-rows, 239
List module, code example excerpt from, 113–114
list of tokens recursive pattern, code example, 240
list processing
  finding the longest string in a list of strings, 281–283
  and recursion depth, 280–284
  in XSLT, 281
list types, as non-atomic simple types, 397
literal result elements, for non-XSLT material, 223
literal string, specifying a value as, 234
loading (reloading) the application, code example of, 328
local-name(*node-set?*), node-set string-valued function, 221
local resource, defined, 62–63
location path expressions. *See* path expressions
location steps
  parts of, 215
  and their components, 215–219
locator elements
  code example for outputting, 244–245
  descriptive content for, 317
locator labels, construction of, 317–318
locator value, for xlink:type for children elements, 62

login page, for application, 322–323
longest verse
  locating with VBScript extension function with MSXML 3, 300–301
  revisited, 297–307
  by tree recursion, 299
  using extension elements and functions, 299–303
longestLoc(), Java extension function, 303–304

M

main() method, of PublishFindDeleteBiz.bat, 481–482
make-row template, XLink application, 247–248
make-rows template, find-neighbors code example for, 249
makePFString.bat file, code example, 438
mapping, the square() function over the list, 280
Mathematical Markup Language (MathML) Version 2.0, release of, 108
max(), Java extension function, 306–307
maxValLoc(), Java extension function, 304–306
measurement code, for time and space for range generation in XSLT, 289–292
Megginson, David
  19 Short Questions about Namespaces (with Answers) by, 75
  development of SAX by, 127
Message Exchange Patterns (MEPs)
  combining SOAP messages to implement, 458
  most common, 428
message instruction element, XSLT processing model, 226
Microsoft IIS/PWS, Web server and ASP processor as part of XSLT setups, 21
Microsoft Internet Explorer (IE). *See* Internet Explorer (IE)
Microsoft MSXML 3.0 (and later), SAX implementation in, 127
Microsoft .NET, as Web services infrastructure, 428
Microsoft SOAP Toolkit, Web site address, 496
.mix PE name, to represent element types from different classes, 112
mixed-content declaration, 90
mixed-content model
  EBNF production and declaration for, 90
  interleaving of elements and, 375–376
mod operator, using to divide items into groups of equal size, 263
mode attribute, using push with, 230–231
modelGroup element, XML Schema, 420
.mod PE, code example, 112
.module PE, code example, 112
modular JSPs, writing, 529
modularity and reuse, 383–392
modularization
  goals and use cases, 110–111
  main tools of, 383
modularization framework
  DTD implementation of, 111–114
  layers, 110
Modularization of XHTML
  deciding if it is worth using, 119
  release of, 108
Mozilla, implementation of XML parsing and CSS styling by, 35
MSXML, and Internet Explorer support for XSLT, 38
MSXML 3.0 (or later), combined XML parser and XSLT processor for XSLT setups, 21
Muench, Steve, grouping problem solution discovered by, 262–263
Multi-Schema Validator (MSV) tool, available from Sun's Web site, 363
multiple nodes, working with, 27
mustUnderstand attribute, in Envelope namespace, 460–461

N

name(*node-set?*), node-set string-valued function, 221
name classes, combining in RELAX NG, 382–383
named templates and recursion, 236–241
NamedNodeMap interface, 175
names and namespaces, 52–60
namespace-alias element, XSLT processing model, 225
namespace axis, XPath, 212

namespace declarations
  including in your code for all processors, 241
  overriding, 58
  scope of, 55–56
Namespace Myths Exploded (Ronald Bourret), 75
namespace nodes, XPath, 210
namespace support
  Node methods for, 174
  in RELAX NG, 377–383
namespace-uri(*node-set?*), node-set string-valued function, 221
namespaced documents
  code example of schema-related markup in, 409–410
  validating, 103–105
namespaces
  approaches needed to have attribute names in, 379
  and attributes, 59–60
  code example of handling of in XSLT and XPath, 57
  code example of suffix redefined in the internal subset of DTD, 104
  default, 58
  and DTDs code example, 104
  and names, 52–60
  and prefixes, 53–55
  in RELAX NG grammar serving as annotations or comments, 379–380
  use of in ContentHandler interface, 135–136
  and well-formed documents, 41–79
namespaces and attributes, RELAX NG, 378–379
Namespaces FAQ (Ronald Bourret), 75
Netscape, implementation of XML parsing and CSS styling by, 35
New Oxford Annotated Bible with the Apocrypha/Deuterocanonical Books, use of for XML data, 196
nextlinks template, find-neighbors code example for, 248–249
NMTOKEN attribute value, 91
NMTOKENS, IDREFS, 91
nns. *See* no namespace (nns)
no namespace (nns), 404
no namespace (nns) document, code example of schema-related markup in, 405–406
node children, Node methods for modifying, 174
Node interface
  DOM (Document Object Model), 170
  and Node types, 172–173
Node methods, overview of, 174
node-set data type, 206
  converting to a string in XPath, 206
node-set functions, 221–222
node-sets
  code example of comparing for equality, 218
  equality and inequality operators with, 217–218
  and how path expressions evaluate to, 214–222
node tests, 215–217
node types
  and default templates, 232
  table of DOM, 173
node types and node properties, summary table of, 210
NodeFilter objects, 184
NodeIterator object
  attaching to a tree, 183
  creating, 183–184
  as traversal interface, 182
NodeList interface, 175
nodes
  precedence rules for, 233
  processing a set of, 27
non-atomic simple types, 397
Notation interface, 173
ns attribute, for specifying the namespace of a name being defined, 377
number() conversion function, in XPath, 206
number data type, in XPath, 206
number element, XSLT processing model, 225
Number Literal, XPath, 207
number-valued node-set functions, 221
numeric types, supported by Jing processor, 393

O

OASIS (Organization for the Advancement of Structured Information Standards), Web site address for, 9
OASIS technical committees, online resources, 518
Object Management Group (OMG), language for specifying CORBA interfaces, 172
OCLC (Online Computer Library Center), 313–315

Office Equipment Web service
  implementation of, 467–479
  writing the SOAP client for, 472–479
official recommendation (R), by W3C, 14
online library science, world epicenter of, 313
online resources, 517–523
  for Java XML processing, 520–521
  for JSP, ASP, and Web applications, 521–522
  OASIS technical committees, 518
  other standards consortia, 518–519
  for software, 522–523
  W3C technical reports, 517–518
  for Web services, 521–522
  for XML and databases, 521
  for XML information, 519–520
Opera, implementation of XML parsing and CSS styling by, 35
or operator, in XPath, 208
order of evaluation, pull and push in XSLT, 226–231
ot.xml file, finding the longest verse in, 283
output element, XSLT processing model, 225

P

param and variable elements, XSLT processing model, 225
parameter entities (PEs)
  common attributes, 112–113
  as documentation, 100–101
  as macros, 100
parameter entity references, use of, 93
parameters
  how they are passed to the stylesheet, 235
  passing from one template to another, 236–241
  position and usage of, 235
parent axis, XPath, 212
Parenthetical expression, XPath, 207
parse() method
  code example with a systemID argument, 167
  code example with an InputSource argument, 168
  using, 131–132
parsed general entities
  internal entity references, 94–95
  syntax for declaring, 94
parser
  creating, 132–134
  creating the JAXP way, 133–134
  creating the SAX2 way, 133
  creating without using JAXP, 142
  defined, 3
  as part of SAX trio, 128
parser attitudes, and well-formed documents, 8–9
parser output, code example for displaying in text area, 158–159
parseRow(), code example for parsing each line, 169
parsers, with attitude, 46–47
ParseURL() function, code example for DOMBuilder's, 189
parsing
  programming languages vs. XML, 9–11
  XML, 7–8, 125–193
  XML vs. programming languages, 9–11
PassAlong.jsp
  code example of receive HTTP request and send SOAP request, 454–455
  code example of receive SOAP request, process, and send to browser,456
path expressions
  abbreviated form of, 219–221
  additional predicates within, 217
  the full form of, 214–219
  and how they evaluate to node-sets, 214–222
  location steps and their components, 215–219
  summary of abbreviations and wildcards, 220
pattern facet
  code example for defining telephone numbers, 399
  and Regular Expressions, 398–399
  using to define a range of integers, 399–400
pattern matching, examples of, 398–399
patterns and grammar, RELAX NG, 371
payload, defined, 458
PCDATA (Parsed Character DATA), using, 7
pdata (personal data) documents, code example for validating, 102
pdata0.dtd, DTD code example, 102
PE naming conventions, 111–112
  within a module, 112–114
PEs (parameter entities). *See* parameter entities (PEs)
PFStringW2J.bat, code example for WSDL to Java, 437
pickverses.jsp, outline form for how it proceeds, 146–151

pipe (|) operator, as choice separator in children-only content model, 89
PIs (processing instructions), in XML documents, 16
plus (+) character, meaning of in EBNF rules, 50
PortType interface, code example, 441
position(), node-set number-valued function, 221
positional axes, 213
Post-Schema-Validation Infoset (PSVI), 92–93
precedence rules, for patterns, 233
preceding axis, XPath, 212
preceding-sibling axis, XPath, 212
predeclared entities, 48–49
prefixes, and namespaces, 53–55
PreparedStatement, 336–337
PrimeFactorsString class, function of, 438
primeFactorsString.htm, the HTML page of, 453–454
primeFactorsString.jsp, code example, 439–440
PrimeFactorsStringPortType interface, function of, 438
PrimeFactorsStringSoapBindingStub class, function of, 438
priority attribute, of xsl:template, 233
processing-instruction element, XSLT processing model, 225
processing instruction nodes, XPath, 210
processing pipeline, for XML+XSLT on the client, 36
programming languages vs. XML, 9–11
proposed recommendations (PRs), by W3C, 14
PRs (proposed recommendations), by W3C, 14
PSVI (Post-Schema-Validation Infoset). *See* Post-Schema-Validation Infoset (PSVI)
public identifiers
  fields, 98
  in SGML, 97–99
pull processing, 226
  code example, 227
push processing, 226
  code example, 227–228
  using with a mode attribute, 230–231

Q

.qname PE name, for dealing with namespace issues, 112
qNames
  in definitions, 378
  technique for validating in documents with namespaces, 103–105
qstring declaration, in XPath expression, 70–71
Query Implementations 1: UPDATE Queries, 339–353
Query Implementations 2: Refset Actions, 353–358
query string, 71
question mark (?) character, meaning of in EBNF rules, 50
$queue parameter, using to hold a list of nodes to visit, 250
quotation marks (“ ”), specifying a value as a literal string with, 234
Quoted String Literal, XPath, 207

R

R (official recommendation), by W3C, 14
Raggett, Dave
  Tidy program by, 47, 265
range() extension function, in Michael Kay’s Saxon, 285
Range interfaces, in DOM Level 2, 171
range measurements 1: namespaces, code example, 290
range measurements 2: limits and the document() function, code example, 290
range measurements 3: table output, code example, 291
rdb, defined, 310
rdbCtl.jsp, as initial source for the control frame, 339
rdbRefSetOps.jsp
  outline of how it proceeds, 354
  working through, 353–358
RDDL (Resource Directory Description Language)
  modules needed to define as an extension of XHTML Basic, 391–392
  standard values for nature and purpose attributes, 78–79
rddl.rng, Resource module, 390–391
RDF (Resource Description Framework), basics of, 311–313
RDF Model and Syntax, W3C recommendation for RDF descriptions, 312
RDF statements, contents of, 312
recursion, via xsl:apply-templates within a variable, 298
recursion depth
  defined, 283

and list processing, 280–284
and stack memory, 283–284
recursive descent, 299
recursive processing, of a list of tokens, 239–241
recursive subjectOptions template, 347
redefine element, XML Schema, 419
ref2link.xsl
main control structure of, 242–243
as part of XLink application, 242–252
refset hashtable, instantiating, 325–326
refset operations
definitions of, 355–357
supported, 353
Regular Expressions
and pattern facets, 398–399
using to define simple types, 399–400
reification, defined, 318
relational database, as XML repository, 309–310
RELAX NG
associating target namespaces with prefixes in, 378
built-in data types, 373
code example of a document with terms and definitions, 376
code example of terms and definitions, 376
combining and replacing definitions in, 383–384
combining multiple definitions of the same name in, 383–384
defining elements and attributes as lists in, 375
and DTD correspondences, 370
facilitating transition from DTDs to, 380–381
features of, 361–362, 369–370
history and current condition, 362–369
local validity assessment by, 403
namespace support, 377–383
namespaces and attributes, 378–379
overview, 369–383
patterns and grammar, 371–373
QNames in definitions, 378
replacing definitions in, 384
tokens and enumerations, 372–373
tools available from Sun's Web site, 363
uniform treatment of elements and attributes, 373–376
validation page, 365
Web site address for the tutorial, 369
and XML Schema, 361–424
vs. XML Schema, 362
vs. XML Schema attitudes to infoset augmentation, 92
RELAX NG grammar
appearance of ns attribute on, 377
vs. DTD (Document Type Definition), 364
for RDDL, 390–392
using a Web application to validate, 365–368
using the command line to validate, 368–369
validating against, 365–369
RELAX NG namespace, element pattern name attribute, 381–383
RELAXNGCC (RELAX NG Compiler Compiler), Web site address, 363
Remote Procedure Call (RPC). *See* RPC (Remote Procedure Call)
remote resource, defined, 63
Request object, code example for processing, parsing, and validating, 367–368
#REQUIRED attribute default, 92
Resource Directory Description Language (RDDL). *See* RDDL (Resource Directory Description Language)
Resource module, in resource.rng, 391–392
resource value, for xlink:type for children elements, 62
resource.rng, code example, 390–391
Result interface, implementation of in verse picker application, 152–153
result tree fragment data type
converting, 234–235
within XSLT, 206
retrieve request, code example, 348
RFC (request for comments), 13
RMI (Remote Method Invocation), 440
root node, of tree, 2
root node type, XPath, 210
root template
code for the form, the table, and the first two rows, 345
code for the okIdStr variable, 344–345
containing HTML tags outputted by, 246
RPC (Remote Procedure Call), 428
representation of in SOAP1.2-2, 466
using SOAP and Web services for, 462–466
Web services invocation of an array version, 432
RPC conventions and XML encoding, 462–466

Ruby Annotation, release of, 108
Russian Doll pattern, defined in Costello's Best Practices, 411

S

SAX (Simple API for XML), 11
- implementation in Java and other languages, 127
- support for, 127
- support for filter writing, 144–145
- treatment of XML documents by, 125–126

SAX application, objects that work closely with each other in, 128–129
SAX callbacks, in ActiveSAXbookpicker.asp, 190–193
SAX filters, 143–157
SAX parser configurations, 136
SAX parsing, for non-XML data, 157–169
SAX programming, basic, 126–143
SAX to DOM conversion, code example, 153
SAX2
- AttributesImpl interface, 136–137
- available on SourceForge, 127
- interface that declares parser object methods, 131–132

Saxon (Michael Kay), use of recursion by, 284
SAXParser class, JAXP, 133–134
SAXParserFactory class, JAXP, 133–134
SAXSource object, creating, 153
SAXversepicker/pickverses.jsp, skeletal form code example, 144–145
scalar data types, in XPath, 199
schema document, code example of namespace-related markup in, 410–411
schema element
- attributes of root, 418–419
- content model of, 418–419

Schema Information Set Contributions, W3C definition XS1R 2.3, 402
schema validation, setting the parser's properties for, 406–407
schema-validity assessment, aspects of, 403
search actions, supported by the application, 320–321
SELECT queries, using, 334–336
self axis, XPath, 212
semantic attributes, 64
semantic markup vs. behavioral markup, 63–64
sequence separator, comma as in children-only content model, 89
serialization format, XML as, 11
server stub
- code example for factors() method, 444–445
- code example for java.net.URL constructor, 443
- code for default constructor, 443
- for Web services, 442–445

servlet code, troubleshooting in JSP, 525–527
sessCache, function of, 326
session cache
- code example for checking for username, 326
- instantiating, 325–326

session initialization, and application (re)loading, 326–328
session variables, code example for initializing, 327
setAppCacheDB(), code example, 335–336
setAppCacheRootElt() function, code example, 330–331
setNamespaceAware() method, of SAXParserFactory, 136
setString(), using to fill in the value of a PreparedStatement parameter, 337
SGML (Standard Generalized Markup Language)
- international standard for, 13
- public identifiers, 97–99
- use of to define HTML, 13

SGML (Standard Generalized Markup Language) documents vs. XML documents, 42
SGML/HTML vs. XML/XHTML, 45–46
sibling nodes, of tree child nodes, 3
Simple API for XML . *See* SAX (Simple API for XML)
Simple Object Access Protocol (SOAP) . *See* SOAP (Simple Object Access Protocol)
simple types, using Regular Expression to define, 399–400
Single-Request-Response MEP, 458
SOAP (Simple Object Access Protocol)
- as bytes, 453–456
- XML-based message format used by Web services, 426

SOAP 1.2
- contents of Part 1, 457
- major sections in Part 2, 457

overview of, 456–462
SOAP as XML
the HTML page, 448–449
SOAP client, 428–429
code example of the PortType interface, 441
JSP code for, 439
SOAP client JSP, part 1, code example, 473–474
SOAP encoding, and the data model, 464–466
SOAP envelope
code example for XSchema for, 460
code example of function for building, 450
SOAP message
convention for passing a procedure call in, 446
exchange model, 458
manufacturing directly from HTML form data, 448–456
parts of, 458
the root element and its schema definition, 459–460
setting up TCPMon for, 475
the XML structure of, 458–462
SOAP node, 458
SOAP request, code example of within HTTP POST request, 446
SOAP response, code example of within HTTP POST request, 447
SOAP returned values, and an XSLT to display them, 451–452
SOAP server, 428–429
converting a Java class to, 434
SOAP specification, 430
SOAP1.2-2, representation of RPC in, 466
SOAP1.2-2, Section 4
changes in, 464–465
values, types, and encoding, 465–466
software, online resources for, 522–523
Source interface, implementation of in verse picker application, 152–153
SourceForge, Web site address, 127
specification map, summarizing the current status of SOAP, WSDL, and UDDI, 429–431
stack memory, and recursion depth, 283
standards
development of by W3C, 13
for XML text and XML tree, 12
start tag, format of, 4–5
startDocument() method, use of in ContentHandler interface, 135
startElement() method
code example for callback to echo input document, 151
defined by content handler, 139–140
use of in ContentHandler interface, 135–136
startElement() method and attributes, function of, 136–137
Statement and ResultSet, 334–336
straight quotes ("), predeclared character reference for, 49
string() conversion function, in XPath, 206
string data type, in XPath, 206
string-valued node-set functions, 221
strip-space and preserve-space elements, XSLT processing model, 225
struct accessors (names), referenced in SOAP1.2-2 Section 4, 465–466
Struct module, reuse of within the Frames module, 387–388
structural node types, 210
stylesheet language, XSLT as, 20
stylesheet languages
a brief history, 30–39
and browsers, 30–39
subject keywords, 310–311
subjectOptions template, code example of straight linear recursion of subject keywords, 347
submission record, definition and code example, 318
submission records, storing in a database, 319–320
submissions, 310–311
submit elements, 318–319
subset use case, XHTML modularization, 110
summary table, from data items to, 267–268
Sun's JAX-RPC, as Web services infrastructure, 428
syntactical diagram, for Hello, XML code listing, 5
syntax, as language component, 2–3
syntax diagrams
an English-language example, 3–4
tree structures formed by, 3
system functions, in XslUtil used in code examples, 294–295
SYSTEM identifiers vs. PUBLIC identifiers, 97–99

T

tables
- outputting into rows and columns, 264
- regrouping with summation by category, 264–265

tables of helpers, code example, 237–238
tables of helpfulness, code example, 236
TagCount with frames, 142
tail recursion vs. linear recursion, 284
TCPMon listener, function of, 445
TCPMon (TCP monitor) utility, and the messages, 445–447
template rule, XSLT processing model, 223–233
templates, precedence rules for, 233
Tennison, Jeni, stylesheet for converting flat to hierarchical structures, 273
testing, a parser's compliance with the XML specification, 47
testmt.dtd, sample lines from, 282
Text module
- Block.class defined as a choice of text containers in, 389
- Block.mix defined in, 388
- Block.model defined in, 388

text nodes, XPath, 210
text-only content model, 90
The World Factbook
- building a hierarchy with keys, 276–278
- categories in, 274
- how divisions are indicated in HTML file structure, 275–276
- the HTML file structure, 275–276
- published by the CIA, 273–275

Tidy HTML-to-XHTML conversion program, using, 266
to-locators, grouping references by using xsl:key and generate-id(), 263
tokens and enumerations, RELAX NG, 372–373
Tomcat. *See* Apache Tomcat
top control page (rdbCtl.jsp), outline of how it proceeds, 325
top-level elements, summarized for XSLT processing model, 225
Transformer object, JAXP, 151–154
TransformerFactory instance, creating, 152
translate() function, code example, 244
transversal interfaces, 182–186
Traversal interfaces, in DOM Level 2, 171
traversal interfaces, navigation basics, 182–183
tree
- conditions, needed for parsing XML, 8
- defined, 2–3
- diagram for Hello, XML code listing, 5

tree address, simple DOM example, 176–181
tree recursion (binary split)
- based on the document tree, 297–299
- longest verse by, 299
- vs. linear recursion, 285–292

tree representations, of XML, 6
treeAddr.jsp
- code of main page, 178–179
- page directive and session variables code, 179

TreeWalker object, as traversal interface, 182
troubleshooting, in JSP, 525–530

U

UCS (Universal Character Set), 206
UDDI (Universal Description, Discovery, and Invocation)
- code example of procedure that reports exceptions, 484
- publish, find, bind with, 479–489
- used by Web services, 426

UDDI exceptions, 484–485
uiCitation.jsp, functionality of, 340–342
uiCitation.xsl, code example of the root and top-level elements of, 344
uninterpreted languages. *See* formal (or uninterpreted) languages
union expressions, formed with | (pipe) operator, 218–219
union type, code example, 397
unique locators, using xsl:key and generate-id(), 261
units of storage, defined, 48
Universal Description, Discovery, and Invocation (UDDI). *See* UDDI (Universal Description, Discovery, and Invocation)
upAddCite.jsp
- code examples, 348–353
- outline of how it proceeds, 347–348

update actions, supported by the application, 321
UPDATE queries
- overview, 340–342
- using, 334–336

updates, Web site address for, 492
upstream parser, for SAX filters, 144–145
URI (Uniform Resource Identifier), types of, 509–510

URL (Uniform Resource Locator), 509–510
URN (Uniform Resource Name), 509–510
U.S. presidents by quarter-century and party affiliation, 270–272
- XHTML to XML, 270
- from XML to XHTML summary table, 271–272

use attribute, possible values for, 421
use cases, goal of modularization to support, 110
user agent, defined, 510
user-interface files
- for the application, 321–322
- defined, 339

UTF (Unicode Transfer Format), 16
UTF-8, encoding supported by XML parsers, 16
UTF-16, encoding supported by XML parsers, 16

V

valid documents
- defined, 82
- and validating parsers, 19

validate.asp program, code example, 85–86
validateJS.js program, code example, 84–85
validaterng.jsp, how it proceeds, 366
validating parsers
- defined, 82
- running, 82–87
- and valid documents, 19

validation and DTDs, 81–124
validation tools, for validating against RELAX NG grammars, 365–369
validom.jsp page, code example, 86–87
value, specifying, 234
value-of instruction element, XSLT processing model, 226
value space, 394
VariableReference expression, XPath, 207
variables, result tree fragments, and node sets, for implementing make-row, 241
VB Book Picker, running the application, 187–188
VBScript extension function
- calling by the root template, 301
- code example of it defined in msxml:script, 301
- with MSXML 3 for locating longest verse, 300–301

vdb, defined, 309–310
vector of DOM trees, as XML repository, 309–310
verse picker application, 145–151
- character() callback for, 146
- code for creating a filter object and parsing, 150
- code for initializing parameters and processing request, 150
- code for root element attribute that show match string, 145–146
- outline form for how it proceeds, 146–151
- URL that outputs verses containing Bethlehem, 145

VersePicker class
- code for creating a filter object and parsing, 154
- converting to text with markup using Transformer, 153–154
- defining, 148–149

VersePicker inputs, code example, 147
$verses parameter, using to hold a list of nodes visited, 250

W

W3C (World Wide Web Consortium)
- development of standards by, 13
- procedures and recommendations, 13–15
- XML recommendations issued by, 1
- XML standards and, 11–15

W3C recommendations, XML Schema Part 1 and 2 designations for, 401
W3C technical reports, online resources, 517–518
WAPForum, Web site address, 98
WDs (working drafts), by W3C, 13–14
Web application
- ASPs, 503–505
- CGI framework, 502–503
- code example for timing a stylesheet, 257–258
- general framework, 501–502
- improvements to backend processing, 503
- prerequisites and setups for hands-on work, 21–22
- using to validate RELAX NG grammars, 365–368

Web application (external) functions vs. extension functions, 296–297
Web application validator, error message from, 84

Web services (WSs), 425–490
  areas of application, 427–428
  components of a service, 428–429
  components of with there places of residence, 433
  creating a string version of an RPC, 432–447
  diagram of components of, 429
  diagram that summarizes the vision, 426–427
  example of implementing a procedure call, 431–447
  features in, 426
  important infrastructures, 428
  the Java Class for, 433–434
  online resources for, 521–522
  and the programmer, 428
  the SOAP client, 440–442
  what they are, 425–431
Web Services Deployment Descriptor language (WSDD), tool for manual deployment of Web services, 470–472
Web Services Description Language (WSDL). *See* WSDL (Web Services Description Language)
Web services examples, installation guide, 499–500
Web Services Inspection Language (WSIL), from IBM and Microsoft, 431
Web Services Interoperability Organization (WSIO), Web site address, 431
Web site address
  for articles about Web services, 428
  Axis User Guide, 471
  for complete list of XHTML modules, 111
  for current version of RELAX NG implementation of XHTML modularization, 384
  for diagram of built-in data types, 393
  for discussion on xml-dev list, 403
  for downloading IBM's Web Services Tool Kit (WSTK), 431
  for the encodingStyle URI for encoding SOAP 1.2 Part 2, 430
  for handlers that can be registered with the parser, 130
  for IBM's Web Services ToolKit (WSTK), 480
  for in-depth treatment of language identifiers in XML, 60
  for information about pull model vs. SAX, 127
  for James Clark's RELAX NG materials, 363
  for James Clark's separate frames DTD discussion, 109
  for Java-XML tutorial, 150
  for Jing data types Web page, 393
  for Jon Bosak's Four Religious Works package, 65
  for latest version of Xerces, 400
  for list of HTML and XHTML differences, 46
  Microsoft SOAP Toolkit, 496
  for OASIS, 9
  for OASIS technical committee, 363
  for RDDL information, 76
  for *RDF Model and Syntax*, 312
  for RELAX NG tutorial, 369
  for RELAXNGCC (RELAX NG Compiler Compiler), 363
  for Robin Cover's XML language identifiers page, 60
  for Roger Costello's Schematron information, 418
  for SourceForge, 127
  for specification on using dcmes elements within RDF descriptions, 313
  for Sun's Developer Connection site, 363
  for Sun's JAX-RPC Web services infrastructure, 428
  for Tidy program for fixing HTML documents, 47
  for top-level readme.htm file in book code archive, 492
  for various XML Namespaces resources, 75
  for W3C recommendations and technical reports, 1
  for WAPForum, 98
  for Web Services Inspection Language (WSIL), 431
  for Web Services Interoperability Organization (WSIO), 431
  for working draft of DOM Level 3 module, 173
  for the *The World Factbook*, 274
  XML Base recommendation, 132
  XML Schema Best Practices (Roger Costello), 401
  for XML Schema built-in data types and fundamental facets values, 395

XML Schema Tutorial (Roger Costello), 401
for XML test suite, 47
for XPointer definition, 66
for XSchema for Envelope, Header, and Body elements, 459
WebSphere, as Web services infrastructure, 428
well-formed documents
and namespaces, 41–79
and parser attitudes, 8–9
well-formedness conditions, and parser attitudes, 8–9
While loop, common pattern of working with a traverser, 183
whitespace facet, possible values for, 395
whitespace-only nodes
code example for removing, 186
eliminating with a tree-walker traversal, 185–186
wildcards for extensibility, in XS1, 423
wireless devices, and XHTML Basic, 109
WML DTD, code example from, 98–99
WML1.2 DTD, declaring synonyms for CDATA, 100–101
working drafts (WDs), by W3C, 13–14
writeRefs() method, refset, 353
WSDD language. *See* Web Services Deployment Descriptor language (WSDD)
WSDL (Web Services Description Language)
access points and interfaces described in, 426
code example for generating descriptions, 435–437
to Java, 437–439
main structural points, 436–437
WSDL specification, as note published by W3C, 430–431
Wsdl2java program, batch file to invoke, 437

X

Xalan. *See also* Apache Xalan
JavaScript extension element with, 302–303
results from running binary-split and linear-recursion algorithms, 288–292
xdb, defined, 310
Xerces. *See also* Apache Xerces
turning off automatic schema validation in, 406
Web site address for latest version, 400
XHTML
evolution of, 109
reason to use instead of HTML, 47–48
XHTML 1.1-Module-Based XHTML, release of, 108
XHTML Basic
release of, 108
and wireless devices, 109
and XHTML modularization, 107–119
XHTML Basic DTD, 114–119
driver file, 114–117
xhtml-basic10.dtd driver file
first part of in XHTML Basic DTD, 115
second part of in XHTML Basic DTD, 116–117
XHTML document
code example of, 44
program for converting an HTML document to, 47
XHTML driver file, for the union of transitional and frames DTDs, 385–386
XHTML DTD, coreattrs PE used in attribute declarations, 100
XHTML modularization
the frames problem, 386–387
RELAX NG implementation of, 384–392
and XHTML Basic, 107–119
XHTML modules, Web site address for complete list of, 111
XHTML specifications, release of with DTD implementations, 108
XHTML table structure, converting from XHTML to XML, 266–268
xhtml.rng, showing nothing but include patterns, 385–386
XLink (XML Linking Language), purpose of, 60
XLink
Bible commentary example, 65–69
code example, 65–69
extended link code example, 65–67
XLink application
code example for constructing XPointers in, 245–246
code example for outputting arcs in, 243
code example for outputting locator element in, 244–245
code example for outputting locators, 244–245
code example for start tag and root template, 243

XLink application *(continued)*
- code example with extended links, 67–69
- the code of, 242–252
- creating and using a linkbase, 196–198

XLink attributes
- summary table of, 64
- used to describe links within XML data, 59
- and XLink graphs, 60–64

XLink graphs
- specifying for extended link element, 62–64
- and XLink attributes, 60–64

XLink module, in xlink.rng, 391–392

XLink simple link element
- vs. an HTML hyperlink, 61–62
- similarities to an HTML hyperlink, 62

xlink:actuate, use of in behavioral markup, 62–64

xlink:arcrole attribute
- use of, 64
- Web site address for use guidelines, 78

xlink:href attribute, required for locator elements, 63

xlink:label attribute, needed for locator and resource elements, 63

xlink:role attribute
- use of, 64
- Web site address for use guidelines, 78

Xlinks, XSLT stylesheet to process, 72–73

xlink:show, use of in behavioral markup, 62–64

xlink:title attribute, use of, 64

xlink:type attribute
- possible values of for children elements, 62
- values, 60

XML (eXtensible Markup Language). *See* XML

XML
- approaches to displaying in a browser, 35
- code examples, 15–20
- components, 2
- converting fixed-width records to, 163–169
- with CSS in a browser, 32
- defined, 2
- displaying on the client, 35–39
- document-oriented vs. data-oriented, 19–20
- the essence and alternative views of, 12
- vs. HTML, 45
- introduction to, 1–39
- as just syntax with no interpretation, 10
- key to success of, 36–37
- parsing, 7–8, 125–193
- procedure for constructing a syntactical tree, 8
- processing pipeline, 37
- vs. programming languages, 9–11
- as a serialization format, 11
- sources and results, 152–153
- structure of a SOAP message, 458–462
- tree representations of, 6
- uses for, 1
- why it is so great, 9

XML 1.0
- EBNF rules/productions, 50–51
- XML foundations set down in, 1

XML 1.0 perspective vs. the XML namespaces perspective, 54

XML and databases, online resources for, 521

XML attribute types, supported by Jing processor, 393

XML attributes
- and a DTD, 17–19
- XML documents with, 17–18

XML Base recommendation, Web site address, 132

XML catalogs, OASIS technical committee work on, 99

XML code examples, without a DTD, 15–17

XML data
- code example for converting hashtable data to, 158–159
- diagram of mutual relationship with parser and application, 11
- the structure of, 310–319

XML database
- common ways to implement, 309
- as XML repository, 309–310

xml-dev list, Web site address for discussion on, 403

XML documents
- with attributes, 17–18
- code example of one without a DTD, 52
- comments and processing instructions, 16
- declaration and encoding, 15–16
- with a DTD, 18

elements of Hello, XML, 4–5
encoding and RPC conventions, 462–466
entering CDATA sections into, 49
everything contained in, 120–121
with external DTDs, 18
framework for XSLT programs, 29–30
kinds of material contained in those without a DTD, 49–50
and markup languages, 4–5
online resources for information, 519–520
purpose of parsing, 125
vs. SGML documents, 42
tags needed for empty elements, 7
valid, 19
without a DTD, 48–52
XML languages
grammar rules of, 6–7
for representation of metadata, 310
XML languages and documents, 2–6
XML Link (XLink), specifying the source and target of, 12
XML name types, supported by Jing processor, 393
XML Namespaces (James Clark), 75
XML Namespaces
a brief list of confusions and controversies, 75–76
controversies and RDDL, 74–79
procedure used to establish a namespace, 53–54
resources list for information about, 75
treatment of attributes and elements by, 59–60
XML foundations set down in, 1
XML Namespaces by Example (Tim Bray), 75
XML notation, for RDF descriptions, 312
XML parsers, 2
error handling by, 47
mediation between XML data and applications by, 10–11
two most common encodings supported by, 16
XML Pointer specification (XPointer), using XPath within, 12
xml: prefix, attributes with, 59, 59–60
XML processor, 2
XML Reader, code example for obtaining, 366
XML reader and error handler, code example for XML Schema, 407–408
XML repository, 309–359
XML Schema, 74
annotations and extending schemas, 418
author choices and best practices, 415
code example of simple type definitions, 395–396
contents of, 417–418
element element content models, 411–413
features of, 361–362
framework for validation, 406–409
group, attributeGroup, modelGroup, 420
import element, 419
include element, 419
mapping the namespace and target namespace to a prefix, 412–413
mechanism for creating user-defined derived types, 393–400
named types variation, 411–413
overview, 417–424
redefine element, 419
and RELAX NG, 361–424
vs. RELAX NG, 362
vs. RELAX NG attitudes to infoset augmentation, 92
simple data type categories, 397–398
simple variations and best practices, 411–417
validation page, 406–407
XML Schema data types
and datatypeLibrary, 392–400
supported by the Jing processor, 393
XML Schema Part 1 (XS1)
code example of parsing and schema validation, 408–409
a simple schema and a validation framework, 404–409
structures, 401–424
validation, assessment, and PSVI, 402–404
W3C definition XS1R 2.1, 403
wildcards for extensibility, 423
XML Schema Part 2 (XS2)
built-in types, 392–393
user-defined types, 394–400
XML schema validation page, input elements, 407
XML specifications
selected list of, 14
test suite for testing a parser’s compliance with, 47

XML standards, and W3C, 11–15
XML syntax, 370–371
xml4css.xml, code example, 31–32
xml:base attribute
defined in XML Base recommendation, 132
purpose and use of, 60
XML+CSS, 35
XMLFilter, Java SAX interface that extends XMLReader, 144–145
XMLFilterImpl class, extending, 144–145
xml:lang attribute, using, 59
XMLReader, defined, 131
XMLReader interface, code example for minimal implementation of, 160
XMLReader methods, other than parse(), 159–161
XML+XSLT, on the client, and more on processing pipelines, 35–37
XML+XSLT=>HTML, on the server, 38
XML+XSLT=>XSL-FO, on the server, 38–39
XPath
attributes in, 25
Boolean operators, 208
document() function, 69–70
expressions other than path expressions, 207–208
language and data model, 198–208
mod operator, 263
namespaces in, 56–57
node types and node properties, 210
primary expressions, 207
three things you will learn while learning, 198
type conversions, 206
XPath and XSLT, code example of namespace handling in, 57
XPath axes, 212–214
XPath data model, and the node-set data type, 208–214
XPath data types
and the context of evaluation, 199–201
and values, 206–208
XPath examples, 200–201
XPath expressions, 25
with explanations, 200–201
vs. XPointer expressions, 67
and XSLT programs, 20–30
XPath filtering, in an Apache-specific way, 357–358
XPath language, and data model, 198–208
XPath node-set, axes and content node position, 214
XPath operators, 208
XPath specification, tree diagram conventions from, 5
XPath tester, entry form for, 201–202
XPath tree
axes of, 211–214
generating a range of numbers as nodes in, 285
for an XML document with attributes and namespaces, 209
XPath Tree Model, for a document with a comment and a PI, 16–17
XPath values and data types, 206–208
XPath within XSLT (pdataNameTable.xsl), code example, 199–200
XPath, XSLT, and XLink processing, 195–253
XPointer expressions
definition of, 66–67
vs. XPath expressions, 67
XPointers
constructing in XLink application, 245–246
extracting from dreams.xml and encoding, 71
xptrans.jsp
coding in ASP, 204–205
evaluating XPath expressions with, 201–205
how the JSP (Java) code proceeds, 202–203
reader parameters expected by, 202
XQuery 1.0: An XML Query Language, being developed by W3C, 310
xs/xsEx1.xml, code example, 401–402
XS1. *See* XML Schema 1
XS1R, 401
Section 2.3 definition of additions to document infoset, 402
Section 2.4 definitions of levels of conformance, 402
XS2. *See* XML Schema 2
XS2R, 401
xsEx1nns.xsd, code example of our first XS1 schema, 404
XSL-FO, 34
XSL language, 33
xsl:apply-imports, for applying an imported template rule, 233
xsl:apply-templates element
as main structure of ref2link.xsl, 242–243
recursion via, 298

xsl:apply-templates instruction
  evaluating, 226–227
  sorting element for, 229–230
xsl:attribute, using to construct an output attribute, 29
xsl:for-each
  sorting element for, 229–230
  using, 27
  using for filtering and mapping in list processing, 281
xsl:for-each instructions, evaluating, 226–227
xsl:import, bringing in material from another stylesheet with, 233
xsl:include, bringing in material from another stylesheet with, 233
xsl:key element
  and the key() function, 259–260
  required attributes, 259
xsl:param, associating a name with a value with, 234
xsl:sort child element, using with xsl:for-each and xsl:apply-templates, 229–230
XSLT (eXtensible Stylesheet Language for Transformations). *See* XSLT
XSLT, 6, 33–34
  algorithms and efficiency, 255–307
  for citation input form, 342–347
  code example for stylesheet with a JavaScript callback, 258–259
  code example of list of tokens recursive pattern, 240
  to display SOAP returned values, 452
  distinct nodes and keys, 259–263
  Internet Explorer and MSXML support for, 38
  list processing in, 280–284
  namespaces in, 56–57
  push and pull contrasted in, 231
  setups for running XSLT programs, 21–22
  specific patterns and timing, 256–259
  table showing time and space for range generation in, 289
  using extension functions within, 293–297
  using to generate large data sets, 284–292
XSLT and XPath, code example of namespace handling in, 57
XSLT parameters, variables, and result tree fragments, 234–235
XSLT processing model, 223–233
  instruction elements summarized, 225–226
  top-level elements summarized, 225
XSLT processor, URL for invoking, 22
XSLT Programmer's Reference (Michael Kay), 195
XSLT programs
  ASP code to run, 23–24
  browser support for, 37–39
  code example for producing an HTML page, 25–30
  code for outer shell of, 30
  with conditional expressions, 28
  for Hello, XML world! listing, 23
  helloXSL/helloSelect.xsl, 26
  improving efficiency of, 256–259
  JSP code to run, 24
  running, 22–25
  as stylesheets, 20
  timing a Web application, 257–259
  and XPath expressions, 20–30
XSLT stylesheet
  invoking external code written in other languages in, 296–297
  to process XLinks, 72–73
XSLT variable, creating, 29
XSLT/XPath extension functions, 71–73
xsl:template, priority attribute, 233
XslUtil
  code example of system functions in, 292
  system functions used in code examples, 294–295
xsl:variable
  associating a name with a value with, 234
  using, 26–27

Y

YACC (Yet Another Compiler Compiler), 363–364