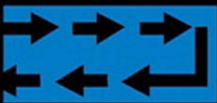
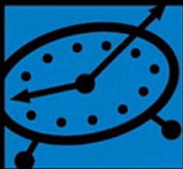


$$\Gamma_L(f) = \frac{Z_L(f)}{Z_C(f)}$$



XML

B Y E X A M P L E



```
<section><xsl:template  
match="/"><HTML><UL><xsl:for-each select="article/section/title"><LI><A><xsl:value-of select="."/></A></LI>  
</UL></section></HTML>  
</xsl:template>  
</xsl:stylesheet>  
</xsl:output/>  
</xsl:transform>
```



XMIL

B Y E X A M P L E

que[®]

201 West 103rd Street
Indianapolis, Indiana 46290

Benoît Marchal

XML by Example

Copyright © 2000 by Que ®

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-7897-2242-9

Library of Congress Catalog Card Number: 99-66449

Printed in the United States of America

First Printing: December 1999

01 00 4

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Que cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Publisher

John Pierce

Acquisitions Editor

Todd Green

Development Editor

Susan Hobbs

Technical Editor

Karl Fast

Managing Editor

Thomas F. Hayes

Project Editor

Karen S. Shields

Copy Editor

Sossity Smith

Indexer

Tina Trettin

Proofreader

Maribeth Echard

Team Coordinator

Julie Otto

Media Developer

Jay Payne

Interior Designer

Karen Ruggles

Cover Designer

Rader Design

Copy Writer

Eric Borgert

Production

Louis Porter Jr.

Contents at a Glance

Introduction	1
1 The XML Galaxy	5
2 The XML Syntax	41
3 XML Schemas	69
4 Namespaces	107
5 XSL Transformation	125
6 XSL Formatting Objects and Cascading Style Sheet	161
7 The Parser and DOM	191
8 Alternative API: SAX	231
9 Writing XML	269
10 Modeling for Flexibility	307
11 N-Tiered Architecture and XML	345
12 Putting It All Together: An e-Commerce Example	381
Appendix A: Crash Course on Java	457
Glossary	485
Index	489

Table of Contents

Introduction	1
The <i>by Example</i> Series	1
Who Should Use This Book	1
This Book's Organization	2
Conventions Used in This Book	3
1 The XML Galaxy	5
Introduction	6
A First Look at XML	8
No Predefined Tags	9
Stricter	10
A First Look at Document Structure	10
Markup Language History	14
Mark-Up	14
Procedural Markup	14
Generic Coding	17
Standard Generalized Markup Language	18
Hypertext Markup Language	20
eXtensible Markup Language	26
Application of XML	28
Document Applications	29
Data Applications	29
Companion Standards	32
XML Namespace	33
Style Sheets	33
DOM and SAX	35
XLink and XPointer	35
XML Software	36
XML Browser	36
XML Editors	37
XML Parsers	37
XSL Processor	37
2 The XML Syntax	41
A First Look at the XML Syntax	42
Getting Started with XML Markup	42
Element's Start and End Tags	44
Names in XML	45
Attributes	46
Empty Element	47
Nesting of Elements	47
Root	48
XML Declaration	49

Advanced Topics	50
Comments	50
Unicode	50
Entities	52
Special Attributes	53
Processing Instructions	53
CDATA Sections	54
Frequently Asked Questions on XML	55
Code Indenting	55
Why the End Tag?	56
XML and Semantic	58
Four Common Errors	59
Forget End Tags	59
Forget That XML Is Case Sensitive	60
Introduce Spaces in the Name of Element	60
Forget the Quotes for Attribute Value	60
XML Editors	60
Three Applications of XML	61
Publishing	62
Business Document Exchange	63
Channel	65
3 XML Schemas	69
The DTD Syntax	70
Element Declaration	71
Element Name	72
Special Keywords	72
The Secret of Plus, Star, and Question Mark	73
The Secret of Comma and Vertical Bar	73
Element Content and Indenting	74
Nonambiguous Model	74
Attributes	75
Document Type Declaration	76
Internal and External Subsets	77
Public Identifiers Format	79
Standalone Documents	79
Why Schemas?	80
Well-Formed and Valid Documents	81
Relationship Between the DTD and the Document	82
Benefits of the DTD	84
Validating the Document	84
Entities and Notations	85
General and Parameter Entities	86
Internal and External Entities	87
Notation	89
Managing Documents with Entities	90

	Conditional Sections	.91
	Designing DTDs	.91
	Main Advantages of Using Existing DTDs	.92
	Designing DTDs from an Object Model	.92
	On Elements Versus Attributes	.96
	Creating the DTD from Scratch	.97
	On Flexibility	.97
	Modeling an XML Document	.100
	Naming of Elements	.103
	A Tool to Help	.104
	New XML Schemas	.104
4	Namespaces	.107
	The Problem Namespaces Solves	.108
	Namespaces	.112
	The Namespace Name	.114
	URIs	.114
	What's in a Name?	.115
	Registering a Domain Name	.116
	Creating a Sensible URL	.117
	URNs	.117
	Scoping	.118
	Namespaces and DTD	.119
	Applications of Namespaces	.120
	XML Style Sheet	.121
	Links	.122
5	XSL Transformation	.125
	Why Styling?	.126
	CSS	.126
	XSL	.126
	XSL	.127
	LotusXSL	.127
	Concepts of XSLT	.128
	Basic XSLT	.128
	Viewing XML in a Browser	.129
	A Simple Style Sheet	.131
	Stylesheet Element	.134
	Template Elements	.134
	Paths	.135
	Matching on Attributes	.136
	Matching Text and Functions	.136
	Deeper in the Tree	.137
	Following the Processor	.138
	Creating Nodes in the Resulting Tree	.140
	Supporting a Different Medium	.141
	Text Conversion	.141
	Customized Views	.144

Where to Apply the Style Sheet	145
Internet Explorer 5.0	145
Changes to the Style Sheet	148
Advanced XSLT	149
Declaring HTML Entities in a Style Sheet	153
Reorganizing the Source Tree	153
Calling a Template	154
Repetitions	154
Using XSLT to Extract Information	155
6 XSL Formatting Objects and Cascading Style Sheet	161
Rendering XML Without HTML	162
The Basics of CSS	163
Simple CSS	164
Comments	166
Selector	166
Priority	167
Properties	168
Flow Objects and Boxes	168
Flow Objects	168
Properties Inheritance	169
Boxes	169
CSS Property Values	172
Length	172
Percentage	173
Color	173
URL	173
Box Properties	174
Display Property	174
Margin Properties	174
Padding Properties	175
Border-Style Properties	175
Border-Width Properties	175
Border Shorthand	175
Text and Font Properties	176
Font Name	176
Font Size	176
Font Style and Weight	177
Text Alignment	177
Text Indent and Line Height	177
Font Shorthand	178
Color and Background Properties	178
Foreground Color	178
Background Color	178
Border Color	178
Background Image	178

Some Advanced Features	179
Child Selector	180
Sibling Selector	181
Attribute Selector	181
Creating Content	182
Importing Style Sheets	182
CSS and XML Editors	182
Text Editor	183
Tree-Based Editor	183
WYSIWYG Editors	184
XSLFO	185
XSLT and CSS	185
XSLFO	187
7 The Parser and DOM	191
What Is a Parser?	191
Parsers	192
Validating and Nonvalidating Parsers	193
The Parser and the Application	193
The Architecture of an XML Program	193
Object-Based Interface	194
Event-Based Interface	196
The Need for Standards	197
Document Object Model	198
Getting Started with DOM	198
A DOM Application	199
DOM Node	202
Document Object	203
Walking the Element Tree	204
Element Object	206
Text Object	206
Managing the State	207
A DOM Application That Maintains the State	208
Attributes	210
NamedNodeMap	217
Attr	217
A Note on Structure	218
Common Errors and How to Solve Them	218
XML Parsers Are Strict	218
Error Messages	219
XSLT Common Errors	220
DOM and Java	220
DOM and IDL	220
A Java Version of the DOM Application	221
Two Major Differences	223
The Parser	224

DOM in Applications	225
Browsers	225
Editors	229
Databases	229
8 Alternative API: SAX	231
Why Another API?	231
Object-Based and Event-Based Interfaces	232
Event-Based Interfaces	233
Why Use Event-Based Interfaces?	236
SAX: The Alternative API	237
Getting Started with SAX	237
Compiling the Example	241
SAX Interfaces and Objects	242
Main SAX Events	242
Parser	242
ParserFactory	243
InputSource	243
DocumentHandler	243
AttributeList	244
Locator	245
DTDHandler	246
EntityResolver	246
ErrorHandler	246
SAXException	246
Maintaining the State	247
A Layered Architecture	260
States	261
Transitions	262
Lessons Learned	265
Flexibility	265
Build for Flexibility	265
Enforce a Structure	266
9 Writing XML	269
The Parser Mirror	269
Modifying a Document with DOM	270
Inserting Nodes	274
Saving As XML	276
DOM Methods to Create and Modify Documents	277
Document	277
Node	277
CharacterData	278
Element	278
Text	279
Creating a New Document with DOM	279
Creating Nodes	281
Creating the Top-Level Element	282

Using DOM to Create Documents	283
Creating Documents Without DOM	283
A Non-DOM Data Structure	288
Writing XML	289
Hiding the Syntax	290
Creating Documents from Non-XML Data Structures	291
Doing Something with the XML Documents	292
Sending the Document to the Server	292
Saving the Document	295
Writing with Flexibility in Mind	296
Supporting Several DTDs with XSLT	296
Calling XSLT	303
Which Structure for the Document?	304
XSLT Versus Custom Functions	304
10 Modeling for Flexibility	307
Structured and Extensible	307
Limiting XML Extensibility	308
Building on XML Extensibility	312
Lessons Learned	321
XLink	323
Simple Links	323
Extended Links	326
XLink and Browsers	327
Signature	327
The Right Level of Abstraction	330
Destructive and Nondestructive Transformations	330
Mark It Up!	334
Avoiding Too Many Options	336
Attributes Versus Elements	339
Using Attributes	340
Using Elements	341
Lessons Learned	342
11 N-Tiered Architecture and XML	345
What Is an N-Tiered Application?	345
Client/Server Applications	346
3-Tiered Applications	347
N-Tiers	348
The XCommerce Application	348
Simplifications	349
Shop	349
XML Server	353
How XML Helps	356
Middleware	356
Common Format	357

XML for the Data Tiers	359
Extensibility	359
Scalability	361
Versatility	365
XML on the Middle Tier	366
Client	372
Server-Side Programming Language	375
Perl	376
JavaScript	376
Python	377
Omnimark	377
Java	377
12 Putting It All Together: An e-Commerce Example	381
Building XCommerce	381
Classpath	381
Configuration File	382
Directories	383
Compiling and Running	383
URLs	384
Database	384
The Middle Tier	386
MerchantCollection	393
Merchant	397
Product	404
Checkout	407
Encapsulating XML Tools	417
The Data Tier	429
Viewer and Editor	444
Appendix A: Crash Course on Java	457
Java in Perspective	457
Server-Side Applications	458
Components of the Server-Side Applications	458
Downloading Java Tools	459
Java Environment	459
XML Components	460
Servlet Engine	460
Your First Java Application	461
Flow of Control	464
Variables	465
Class	465
Creating Objects	466
Accessing Fields and Methods	466
Static	466
Method and Parameters	467
Constructors	467
Package	468

Imports	.468
Access Control	.468
Comments and Javadoc	.469
Exception	.470
Servlets	.472
Your First Servlet	.473
Inheritance	.476
doGet()	.477
More Java Language Concepts	.478
This and Super	.478
Interfaces and Multiple Inheritance	.479
Understanding the Classpath	.480
JAR Files	.481
Java Core API	.482
Glossary	.485
Index	.489

Dedication

To Pascale for her never-failing trust and patience.

Acknowledgments

This book is an important station on a long journey. I would like to thank all the people who have helped me and trusted me along the way. In chronological order, Ph. Capelle, who helped a confused student; Ph. van Bastelaer and J. Berge, who were curious about SGML; H. Karunaratne and K. Kaur and the folks at Sitpro, who showed me London; S. Vincent, who suggested I get serious about writing; V. D'Haeyere, who taught me everything about the Internet; Ph. Vanhoolandt, who published my first article; M. Gonzalez, N. Hada, T. Nakamura, and the folks at Digital Cats, who published my first U.S. papers; S. McLoughlin, who helps with the newsletter; and T. Green, who trusted me with this book.

Thanks the XML/EDI Group and, in particular, M. Bryan, A. Kotok, B. Peat, and D. Webber.

Special thanks to my mother for making me curious.

Writing a book is a demanding task, both for a business and for a family. Thanks to my customers for understanding and patience when I was late. Special thanks to Pascale for not only showing understanding, but also for encouraging me!

About the Author

Benoît Marchal runs the consulting company, Pineapplesoft, which specializes in Internet applications, particularly e-commerce, XML, and Java. He has worked with major players in Internet development such as Netscape and EarthWeb, and is a regular contributor to `developer.com` and other Internet publications.

In 1997, he cofounded the XML/EDI Group, a think tank that promotes the use of XML in e-commerce applications. Benoît frequently leads corporate training on XML and other Internet technologies. You can reach him at `bmarchal@pineapplesoft.com`.

Tell Us What You Think!

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

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

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

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

Fax: 317-581-4666
Email: que.programming@macmillanusa.com
Mail: John Pierce
Publisher
Que-Programming
201 West 103rd Street
Indianapolis, IN 46290 USA

This page intentionally left blank

Introduction

The *by Example* Series

How does the *by Example* series make you a better programmer? The *by Example* series teaches programming using the best method possible. After a concept is introduced, you'll see one or more examples of that concept in use. The text acts as a mentor by figuratively looking over your shoulder and showing you new ways to use the concepts you just learned. The examples are numerous. While the material is still fresh, you see example after example demonstrating the way you use the material you just learned.

The philosophy of the *by Example* series is simple: The best way to teach computer programming is using multiple examples. Command descriptions, format syntax, and language references are not enough to teach a newcomer a programming language. Only by looking at many examples in which new commands are immediately used and by running sample programs can programming students get more than just a feel for the language.

Who Should Use This Book

XML by Example is intended for people with some basic HTML coding experience. If you can write a simple HTML page and if you know the main tags (such as `<P>`, `<TITLE>`, `<H1>`), you know enough HTML to understand this book. You don't need to be an expert, however.

Some advanced techniques introduced in the second half of the book (Chapter 7 and later) require experience with scripting and JavaScript. You need to understand loops, variables, functions, and objects for these chapters. Remember these are advanced techniques, so even if you are not yet a JavaScript wizard, you can pick up many valuable techniques in the book.

This book is for you if one of the following statements is true:

- You are an HTML whiz and want to move to the next level in Internet publishing.
- You publish a large or dynamic document base on the Web, on CD-ROM, in print, or by using a combination of these media, and you have heard XML can simplify your publishing efforts.
- You are a Web developer, so you know Java, JavaScript, or CGI inside out, and you have heard that XML is simple and enables you to do many cool things.

- You are active in electronic commerce or in EDI and you want to learn what XML has to offer to your specialty.
- You use software from Microsoft, IBM, Oracle, Corel, Sun, or any of the other hundreds of companies that have added XML to their products, and you need to understand how to make the best of it.

You don't need to know anything about SGML (a precursor to XML) to understand *XML by Example*. You don't need to limit yourself to publishing; *XML by Example* introduces you to all applications of XML, including publishing and nonpublishing applications.

This Book's Organization

This book teaches you about XML, the eXtensible Markup Language. XML is a new markup language developed to overcome limitations in HTML.

XML exists because HTML was successful. Therefore, XML incorporates many successful features of HTML. XML also exists because HTML could not live up to new demands. Therefore, XML breaks new ground when it is appropriate.

This book takes a hands-on approach to XML. Ideas and concepts are introduced through real-world examples so that you not only read about the concepts but also see them applied. With the examples, you immediately see the benefits and the costs associated with XML.

As you will see, there are two classes of applications for XML: publishing and data exchange. Data exchange applications include most electronic commerce applications. This book draws most of its examples from data exchange applications because they are currently the most popular. However, it also includes a very comprehensive example of Web site publishing.

I made some assumptions about you. I suppose you are familiar with the Web, insofar as you can read, understand, and write basic HTML pages as well as read and understand a simple JavaScript application. You don't have to be a master at HTML to learn XML. Nor do you need to be a guru of JavaScript.

Most of the code in this book is based on XML and XML style sheets. When programming was required, I used JavaScript as often as possible. JavaScript, however, was not appropriate for the final example so I turned to Java.

You don't need to know Java to understand this book, however, because there is very little Java involved (again, most of the code in the final example is XML). Appendix A, "Crash Course on Java," will teach you just enough Java to understand the examples.

Conventions Used in This Book



EXAMPLE

Examples are identified by the icon shown at the left of this sentence:

Listing and code appears in monospace font, such as

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

NOTE

Special notes augment the material you read in each chapter. These notes clarify concepts and procedures.

TIP

You'll find numerous tips offering shortcuts and solutions to common problems.

CAUTION

The cautions warn you about pitfalls that sometimes appear when programming in XML. Reading the caution sections will save you time and trouble.

What's Next

XML was introduced to overcome the limitations of HTML. Although the two will likely coexist in the foreseeable future, the importance of XML will only increase. It is important that you learn the benefits and limitations of XML so that you can prepare for the evolution.

Please visit the *by Example* Web site for code examples or additional material associated with this book:

```
<http://www.quecorp.com/series/by\_example/>
```

Turn to the next page and begin learning XML by examples today!

XML Schemas

In Chapter 2, “The XML Syntax,” you learned how to write and read XML documents. More importantly, you learned that XML emphasizes the structure of documents.

This chapter further develops that theme by looking at the DTD, short for *Document Type Definition*, a mechanism to describe the structure of documents. Specifically, you will learn how to

- model XML documents
- express the model in a DTD
- validate a document against its model

The DTD is the original modeling language or schema for XML. However, for historical reasons, the DTD is somewhat limited and people are looking for solutions to overcome these limitations. The W3C is working on an alternative to DTD. We will review the current status of that effort.

This chapter is probably the most abstract chapter in this book. You might want to temporarily skip the second half (starting from the section “Entities and Notations”) and revisit it after you have read through the book.

The DTD Syntax



EXAMPLE

The syntax for DTDs is different from the syntax for XML documents. Listing 3.1 is the address book introduced in Chapter 2 but with one difference: It has a new `<!DOCTYPE>` statement. The new statement is introduced in the section “Document Type Declaration.” For now, it suffices to say that it links the document file to the DTD file. Listing 3.2 is its DTD.

Listing 3.1: An Address Book in XML

```
<?xml version="1.0"?>
<!DOCTYPE address-book SYSTEM "address-book.dtd">
<!-- loosely inspired by vCard 3.0 -->
<address-book>
  <entry>
    <name>John Doe</name>
    <address>
      <street>34 Fountain Square Plaza</street>
      <region>OH</region>
      <postal-code>45202</postal-code>
      <locality>Cincinnati</locality>
      <country>US</country>
    </address>
    <tel preferred="true">513-555-8889</tel>
    <tel>513-555-7098</tel>
    <email href="mailto:jdoe@emailaholic.com"/>
  </entry>
  <entry>
    <name><fname>Jack</fname><lname>Smith</lname></name>
    <tel>513-555-3465</tel>
    <email href="mailto:jsmith@emailaholic.com"/>
  </entry>
</address-book>
```

Listing 3.2: The DTD for the Address Book

```
<!-- top-level element, the address book
      is a list of entries          -->
<!ELEMENT address-book (entry+)>

<!-- an entry is a name followed by
      addresses, phone numbers, etc. -->
```



```

<!ELEMENT entry (name,address*,tel*,fax*,email*)>

<!-- name is made of string, first name
      and last name. This is a very flexible
           model to accommodate exotic name -->
<!ELEMENT name (#PCDATA | fname | lname)*>
<!ELEMENT fname (#PCDATA)>
<!ELEMENT lname (#PCDATA)>

<!-- definition of the address structure
      if several addresses, the preferred
           attribute signals the "default" one -->
<!ELEMENT address (street,region?,postal-code,locality,country)>
<!ATTLIST address preferred (true | false) "false">
<!ELEMENT street (#PCDATA)>
<!ELEMENT region (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
<!ELEMENT locality (#PCDATA)>
<!ELEMENT country (#PCDATA)>

<!-- phone, fax and email, same preferred
      attribute as address -->
<!ELEMENT tel (#PCDATA)>
<!ATTLIST tel preferred (true | false) "false">
<!ELEMENT fax (#PCDATA)>
<!ATTLIST fax preferred (true | false) "false">
<!ELEMENT email EMPTY>
<!ATTLIST email href CDATA #REQUIRED
                preferred (true | false) "false">

```

Element Declaration

1. DTD is a mechanism to describe every object (element, attribute, and so on) that can appear in the document, starting with elements. The following is an example of element declaration:

```
<!ELEMENT address-book (entry+)>
```

After the `<!ELEMENT` markup comes the element name followed by its *content model*. The element declaration is terminated with a right angle bracket.



EXAMPLE

Element declarations are easy to read: The right side (the content model) defines the left side (the element name). In other words, the content model lists the children that are acceptable in the element.

The previous declaration means that an `address-book` element contains one or more entry elements. `address-book` is on the left side, `entry` on the right. The plus sign after `entry` means there can be more than one entry element.



EXAMPLE

2. Parentheses are used to group elements in the content model, as in the following example:

```
<!ELEMENT name (lname, (fname | title))>
```

Element Name

As we saw in Chapter 2, XML names must follow certain rules. Specifically, names must start with either a letter or a limited set of punctuation characters (“_”, “:”). The rest of the name can consist of the same characters plus letters, digits and new punctuation characters (“.”, “-”). Spaces are not allowed in names.

Names cannot start with the string “xml,” and as we will see in Chapter 4, “Namespaces,” the colon plays a special role so it is advised you don’t use it.

Special Keywords

For most elements, the content model is a list of elements. It also can be one of the following keywords:

- `#PCDATA` stands for parsed character data and means the element can contain text. `#PCDATA` is often (but not always) used for leaf elements. Leaf elements are elements that have no child elements.
- `EMPTY` means the element is an empty element. `EMPTY` always indicates a leaf element.
- `ANY` means the element can contain any other element declared in the DTD. This is seldom used because it carries almost no structure information. `ANY` is sometimes used during the development of a DTD, before a precise rule has been written. Note that the elements must be declared in the DTD.

Element contents that have `#PCDATA` are said to be *mixed content*. Element contents that contain only elements are said to be *element content*. In Listing 3.2, `tel` is a leaf element that contains only text while `email` is an empty element:



EXAMPLE

```
<!ELEMENT tel    (#PCDATA)>
<!ELEMENT email EMPTY>
```

Note that `CDATA` sections appear anywhere `#PCDATA` appears.

The Secret of Plus, Star, and Question Mark

The plus (“+”), star (“*”), and question mark (“?”) characters in the element content are *occurrence indicators*. They indicate whether and how elements in the child list can repeat.

- An element followed by no occurrence indicator must appear once and only once in the element being defined.
- An element followed by a “+” character must appear one or several times in the element being defined. The element can repeat.
- An element followed by a “*” character can appear zero or more times in the element being defined. The element is optional but, if it is included, it can repeat indefinitely.
- An element followed by a “?” character can appear once or not at all in the element being defined. It indicates the element is optional and, if included, cannot repeat.



EXAMPLE

The entry and name elements have content model that uses an occurrence indicator:

```
<!ELEMENT entry      (name,address*,tel*,fax*,email*)>
<!ELEMENT address   (street,region?,postal-code,locality,country)>
```

Acceptable children for the entry are name, address, tel, fax, and email. Except for name, these children are optional and can repeat.

Acceptable children for address are street, region, postal-code, locality, and country. None of the children can repeat but the region is optional.

The Secret of Comma and Vertical Bar

The comma (“,”) and vertical bar (“|”) characters are *connectors*.

Connectors separate the children in the content model, they indicate the order in which the children can appear. The connectors are

- the “,” character, which means both elements on the right and the left of the comma must appear in the same order in the document.
- the “|” character, which means that only one of the elements on the left or the right of the vertical bar must appear in the document.



EXAMPLE

The name and address elements are good examples of connectors.

```
<!ELEMENT name      (#PCDATA | fname | lname)*>
<!ELEMENT address   (street,region?,postal-code,locality,country)>
```

Acceptable children for name are #PCDATA or a fname element or a lname element. Note that it is one or the other. However, the whole model can repeat thanks to the “*” occurrence indicator.

Acceptable children for address are street, region, postal-code, locality, and country, in exactly that order.

The various components of mixed content must always be separated by a “|” and the model must repeat. The following definition is incorrect:

```
<!ELEMENT name (#PCDATA, fname, lname)>
```

It must be

```
<!ELEMENT name (#PCDATA | fname | lname)*>
```

Element Content and Indenting

In the previous chapter, you learned that the XML application ignores indenting in most cases. Here again, a DTD can help.

If a DTD is associated with the document, then the XML processor knows that spaces in an element that has element content must indent (because the element has element content, it cannot contain any text). The XML processor can label the spaces as *ignorable whitespaces*. This is a very powerful hint to the application that the spaces are indenting.

Nonambiguous Model

The content model must be *deterministic* or unambiguous. In plain English, it means that it is possible to decide which part of the model applies to the current element by looking only at the current element.



EXAMPLE

For example, the following model is not acceptable:

```
<!ELEMENT cover ((title, author) | (title, subtitle))>
```

because when the XML processor is reading the element

```
<title>XML by Example</title>
```

in

```
<cover><title>XML by Example</title><author>Benoît Marchal</author></cover>
```

it cannot decide whether the title element is part of (title, author) or of (title, subtitle) by looking at title only. To decide that title is part of (title, author), it needs to look past title to the author element.

In most cases, however, it is possible to reorganize the document so that the model becomes acceptable:

```
<!ELEMENT cover (title, (author | subtitle))>
```

Now when the processor sees title, it knows where it fits in the model.

Attributes



EXAMPLE

Attributes also must be declared in the DTD. Element attributes are declared with the `ATTLIST` declaration, for example:

```
<!ATTLIST tel preferred (true | false) "false">
```

The various components in this declaration are the markup (`<!ATTLIST`), the element name (`tel`), the attribute name (`preferred`), the attribute type (`(true | false)`), a default value (`"false"`), and the right angle bracket.



EXAMPLE

For elements that have more than one attribute, you can group the declarations. For example, `email` has two attributes:

```
<!ATTLIST email href CDATA #REQUIRED
                preferred (true | false) "false">
```

Attribute declaration can appear anywhere in the DTD. For readability, it is best to list attributes immediately after the element declaration.

CAUTION

If used in a valid document, the special attributes `xml:space` and `xml:lang` must be declared as

```
xml:space (default|preserve) "preserve"
```

```
xml:lang NMTOKEN #IMPLIED
```



EXAMPLE

The DTD provides more control over the content of attributes than over the content of elements. Attributes are broadly divided into three categories:

- string attributes contain text, for example:

```
<!ATTLIST email href CDATA #REQUIRED>
```
- tokenized attributes have constraints on the content of the attribute, for example:

```
<!ATTLIST entry id ID #IMPLIED>
```
- enumerated-type attributes accept one value in a list, for example:

```
<!ATTLIST entry preferred (true | false) "false">
```

Attribute types can take any of the following values:

- `CDATA` for string attributes.
- `ID` for identifier. An identifier is a name that is unique in the document.
- `IDREF` must be the value of an `ID` used elsewhere in the same document. `IDREF` is used to create links within a document.
- `IDREFS` is a list of `IDREF` separated by spaces.

- ENTITY must be the name of an external entity; this is how you assign an external entity to an attribute.
- ENTITIES is a list of ENTITY separated by spaces.
- NMTOKEN is essentially a word without spaces.
- NMTOKENS is a list of NMTOKEN separated by spaces.
- Enumerated-type list is a closed list of nmtokens separated by '|', the value has to be one of the nmtokens. The list of tokens can further be limited to NOTATIONS (introduced in the section “Notation,” later in this chapter).

Optionally, the DTD can specify a default value for the attribute. If the document does not include the attribute, it is assumed to have the default value. The default value can take one of the four following values:

- #REQUIRED means that a value must be provided in the document
- #IMPLIED means that if no value is provided, the application must use its own default
- #FIXED followed by a value means that attribute value must be the value declared in the DTD
- A literal value means that the attribute will take this value if no value is given in the document.

**EXAMPLE****NOTE**

Information that remains constant between documents is an ideal candidate for #FIXED attributes. For example, if prices are always given in dollars, you could declare a price element as

```
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency NMTOKEN #FIXED "usd">
```

When the application reads

```
<price>19.99</price>
```

in a document, it appears as though it reads

```
<price currency="usd">19.99</price>
```

The application has received additional information but it didn't require additional markup in the document!

Document Type Declaration**EXAMPLE**

The *document type declaration* attaches a DTD to a document. Don't confuse the document type declaration with the document type definition (DTD). The document type declaration has the form:

```
<!DOCTYPE address-book SYSTEM "address-book.dtd">
```

It consists of markup (<!DOCTYPE), the name of the top-level element (address-book), the DTD (SYSTEM “address-book.dtd”) and a right angle bracket. As Listing 3.1 illustrates, the document type declaration appears at the beginning of the XML document, after the XML declaration.



EXAMPLE

The top-level element of the document is selected in the declaration. Therefore, it is possible to create a document starting with any element in the DTD. Listing 3.3 has the same DTD as Listing 3.1, but its top-level element is an entry.

Listing 3.3: An Entry

```
<?xml version="1.0"?>
<!DOCTYPE entry SYSTEM "address-book.dtd">
<entry>
  <name>John Doe</name>
  <address>
    <street>34 Fountain Square Plaza</street>
    <region>OH</region>
    <postal-code>45202</postal-code>
    <locality>Cincinnati</locality>
    <country>US</country>
  </address>
  <tel preferred="true">513-555-8889</tel>
  <tel>513-555-7098</tel>
  <email href="mailto:jdoe@emailaholic.com" />
</entry>
```

Internal and External Subsets

The DTD is divided into internal and external subsets. As the name implies, the internal subset is inserted in the document itself, whereas the external subset points to an external entity.

The internal and the external subsets have different rules for parameter entities. The differences are explained in the section “General and Parameter Entities,” later in this chapter.

The internal subset of the DTD is included between brackets in the document type declaration. The external subset is stored in a separate entity and referenced from the document type declaration.

**EXAMPLE**

The internal subset of a DTD is stored in the document, specifically in the document type declaration, as in

```
<!DOCTYPE address [
<!ELEMENT address (street,region?,postal-code,locality,country)>
<!ATTLIST address preferred (true | false) "false">

<!ELEMENT street (#PCDATA)>
<!ELEMENT region (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
<!ELEMENT locality (#PCDATA)>
<!ELEMENT country (#PCDATA)>
]>
```

**EXAMPLE**

The external subset is not stored in the document. It is referenced from the document type declaration through an identifier as in the following examples:

```
<!DOCTYPE address-book SYSTEM "http://www.xmlli.com/dtd/address-book.dtd">

<!DOCTYPE address-book PUBLIC "-//Pineapplesoft//Address Book//EN"
  "http://catwoman.pineapplesoft.com/dtd/address-book.dtd">

<!DOCTYPE address-book SYSTEM "../dtds/address-book.dtd">
```

There are two types of identifiers: *system identifiers* and *public identifiers*. A keyword, respectively **SYSTEM** and **PUBLIC**, indicates the type of identifier.

- A system identifier is a *Universal Resource Identifier* (URI) pointing to the DTD. URI is a superset of URLs. For all practical purposes, a URI is a URL.
- In addition to the system identifier, the DTD identifier might include a public identifier. A public identifier points to a DTD recorded with the ISO according to the rules of ISO 9070. Note that a system identifier must follow the public identifier.

The system identifier is easy to understand. The XML processor must download the document from the URI.

**EXAMPLE**

Public identifiers are used to manage local copies of DTDs. The XML processor maintains a catalog file that lists public identifiers and their associated URIs. The processor will use these URIs instead of the system identifier.

Obviously, if the URIs in the catalog point to local copies of the DTD, the XML processor saves some downloads.

Listing 3.4 is an example of a catalog file.

Listing 3.4: A Catalog File

```
<XMLCatalog>
  <Base HRef="http://catwoman.pineapplesoft.com/dtd/" />
  <Map PublicId="-//Pineapplesoft//Address Book//EN"
      HRef="address-book.dtd" />
  <Map PublicId="-//Pineapplesoft//Article//EN"
      HRef="article.dtd" />
  <Map PublicId="-//Pineapplesoft//Simple Order//EN"
      HRef="order.dtd" />
  <Extend Href="http://www.w3.org/xcatalog/mastercat.xml" />
</XMLCatalog>
```



EXAMPLE

Finally, note that a document can have both an internal and an external subset as in

```
<!DOCTYPE address SYSTEM "address-content.dtd" [
  <!ELEMENT address (street,region?,postal-code,locality,country)>
  <!ATTLIST address preferred (true | false) "false">
]>
```



EXAMPLE

Public Identifiers Format

The following public identifiers point to the address book:

```
"-//Pineapplesoft//Address Book//EN"
```

There are four parts, separated by “//”:

- The first character is + if the organization is registered with ISO, - otherwise (most frequent).
- The second part is the owner of the DTD.
- The third part is the description of the DTD; spaces are allowed.
- The final part is the language (EN for English).

Standalone Documents

As you have seen, the DTD not only describes the document, but it can affect how the application reads the document. Specifically, default and fixed attribute values will add information to the document. Entities, which are also declared in the DTD, modify the document.

If all the entries that can influence the document are in the internal subset of the DTD, the document is said to be standalone. In other words, an XML processor does not need to download external entities to access all the information (it might have to download external entities to validate the document but that does not impact the content).

Conversely, if default attribute values or entities are declared in the external subset of the document, then the XML processor has to read the external subset, which might involve downloading more files.

**EXAMPLE**

Obviously, a standalone document is more efficient for communication over a network because only one file needs to be downloaded. The XML declaration has an attribute, `standalone`, that declares whether the document is a standalone document or not. It accepts only two values: `yes` and `no`. The default is `no`.

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

Note that a standalone document might have an external DTD subset but the external subset cannot modify how the application reads the document. Specifically, the external subset cannot

- declare entities
- declare default attribute values
- declare element content if the elements include spaces, such as for indenting. The last rule is the easiest to break but it is logical: If the DTD declares element content, then the processor reports indenting as ignorable whitespaces; otherwise, it reports as normal whitespaces.

Why Schemas?

Why do we need DTDs or schemas in XML? There is a potential conflict between flexibility and ease of use. As a rule, more flexible solutions are more difficult, if only because you have to work your way through the options. Specific solutions might also be optimized for certain tasks.

Let's compare a closed solution, HTML, with an open one such as XML. Both can be used to publish documents on the Web (XML serves many other purposes as well). HTML has a fixed set of elements and software can be highly optimized for it. For example, HTML editors offer templates, powerful visual editing, image editing, document preview, and more.

XML, on the other hand, is a flexible solution. It does not define elements but lets you, the developer, define the structure you need. Therefore, XML editors must accept any document structure. There are very little opportunities to optimize the XML editors because, by definition, they must be as generic as XML is. HTML, the close solution, has an edge here.

The DTD is an attempt to bridge that gap. DTD is a formal description of the document. Software tools can read it and learn about the document structure. Consequently, the tools can adapt themselves to better support the document structure.

For example, some XML editors use DTDs to populate their element lists as well as adopt default styling, based on the DTD. Finally, these XML editors will guide the author by making certain the structure is followed.

In other words, the editor is a generic tool that accepts any XML document, but it is configured for a specific application (read specific structure) through the DTD.



EXAMPLE

Figure 3.1 is a screenshot from a DTD-aware editor. Notice that the editor prompts for elements based on the structure.

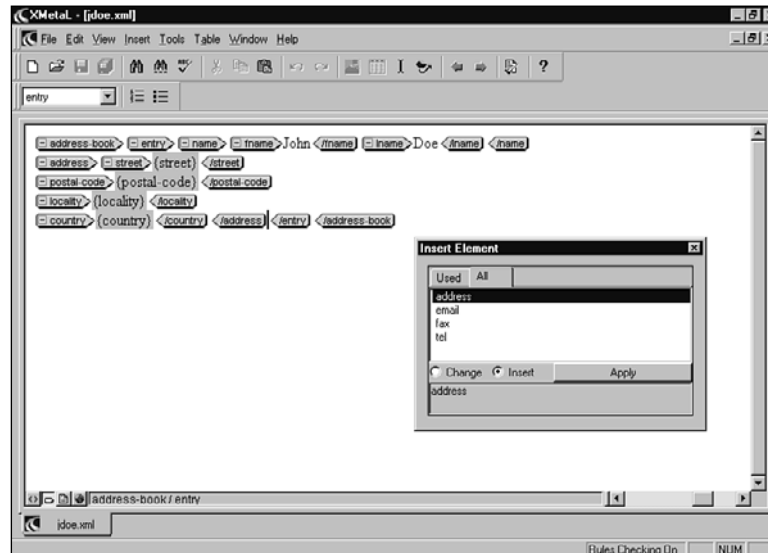


Figure 3.1: XML editor uses the DTD to guide the user.

Well-Formed and Valid Documents

XML recognizes two classes of documents: *well-formed* and *valid*. The documents in Chapter 2 were well-formed, which in XML jargon means they follow the XML syntax. Well-formed documents have the right mix of start and end tags, attributes are properly quoted, entities are acceptable, character sets are properly used, and so on.

Well-formed documents have no DTD, so the XML processor cannot check their structure. It only checks that they follow the syntax rules.

Valid documents are stricter. They not only follow the syntax rules, they also comply with a specific structure, as described in a DTD.

Valid documents have a DTD. The XML processor will check that the documents are syntactically correct but it also ensures they follow the structure described in the DTD.

Why two classes of documents? Why not have only valid documents? In practice, some applications don't need a DTD. Also, among those applications that do, they need the DTD only at specific steps in the process.

The DTD is useful during document creation, when it makes sense to enforce the document structure. However, it is less useful after the creation. For example, in most cases, it is useless to distribute the DTD with the document. Indeed, a reader cannot fix errors in the structure of a document (that's the role of the author and editor), so what is a reader to do with the DTD?

Relationship Between the DTD and the Document

Unless it's overlooked, let me stress the relationship between the DTD and the XML document. The role of the DTD is to specify which elements are allowed where in the document.



EXAMPLE

The documents in Listings 3.6 and 3.7 are valid and respect the DTD in Listing 3.5. The document in Listing 3.6 has a `region` element, whereas the one in Listing 3.7 has none. It works because `region` is a conditional element in the DTD.

Listing 3.5: The DTD

```
<!ELEMENT address (street,region?,postal-code,locality,country)>
<!ATTLIST address preferred (true | false) "false">

<!ELEMENT street (#PCDATA)>
<!ELEMENT region (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
<!ELEMENT locality (#PCDATA)>
<!ELEMENT country (#PCDATA)>
```

Listing 3.6: A Valid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <street>34 Fountain Square Plaza</street>
  <region>0H</region>
```

```
<postal-code>45202</postal-code>
<locality>Cincinnati</locality>
<country>US</country>
</address>
```

Listing 3.7: Another Valid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <street>Rue du Lombard 345</street>
  <postal-code>5000</postal-code>
  <locality>Namur</locality>
  <country>Belgium</country>
</address>
```

However, Listings 3.8 and 3.9 are not valid documents. Listing 3.8 is missing a country element and country is not optional. In Listing 3.9, the region element has a code attribute that is not declared in the DTD.

Listing 3.8: An Invalid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <street>34 Fountain Square Plaza</street>
  <region>OH</region>
  <postal-code>45202</postal-code>
  <locality>Cincinnati</locality>
</address>
```

Listing 3.9: An Invalid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <street>34 Fountain Square Plaza</street>
  <region code="OH">Ohio</region>
  <postal-code>45202</postal-code>
  <locality>Cincinnati</locality>
  <country>US</country>
</address>
```

Another way to look at the relationship between DTD and document is to say that the DTD describes the tree that is acceptable for the document. Figure 3.2 shows the tree described by the DTD in Listing 3.5.



Figure 3.2: The tree for the address

Benefits of the DTD

The main benefits of using a DTD are

- The XML processor enforces the structure, as defined in the DTD.
- The application accesses the document structure, such as to populate an element list.
- The DTD gives hints to the XML processor—that is, it helps separate indenting from content.
- The DTD can declare default or fixed values for attributes. This might result in a smaller document.

Validating the Document

You can validate documents with an XML processor. I invite you to download XML for Java from the IBM Web site at www.alphaworks.ibm.com. There are other XML processors, but I will use the IBM one in Chapter 5, “XSL Transformation,” and Chapter 8, “Alternative API: SAX.”

XML for Java is a Java application. You don’t need to be a Java programmer to use it, but you must have installed a Java runtime on your system. You can download a Java runtime from java.sun.com.

Tools are sometimes updated. If the status of XML for Java changes, we will post an update on the Macmillan Web site at www.quecorp.com/series/by_example. If you experience a problem finding the tool, visit www.quecorp.com/series/by_example.



EXAMPLE

The XML for Java comes with a command-line version that you can use to validate documents against their DTD. To validate the document in Listing 3.1, save it in a file called “abook.xml,” save its DTD in the file called “address-book.dtd,” and issue the command:

```
java -classpath c:\xml14j\xml14j.jar;c:\xml14j\xml14jsamples.jar
  ➔ XJParse -p com.ibm.xml.parsers.ValidatingSAXParser abook.xml
```

This looks like a long and complex command line. If you are curious, Appendix A breaks it into smaller pieces.



OUTPUT

This command assumes XML for Java is installed in the `c:\xml14j` directory. You might have to update the classpath for your system. If everything goes well, the result is a message similar to

abook.xml: 1420 ms (24 elems, 9 attrs, 105 spaces, 97 chars)

If the document contains errors (either syntax errors or it does not respect the structure outlined in the DTD), you will have an error message.

CAUTION

The IBM for Java processor won't work unless you have installed a Java runtime.

If there is an error message similar to "Exception in thread "main" java.lang.NoClassDefFoundError," it means that either the `classpath` is incorrect (make sure it points to the right directory) or that you typed an incorrect class name for XML for Java (XJParser and `com.ibm.xml.parsers.ValidatingSAXParser`).

If there is an error message similar to "Exception in thread "main" java.io.FileNotFoundException: d:\xml\abook.xml", it means that the filename is incorrect (in this case, it points to "abook.xml" instead of "abook.xml").

TIP

You can save some typing with batch files (under Windows) or shell scripts (under UNIX). Adapt the path to your system, replace the filename (abook.xml) with "%1" and save in a file called "validate.bat". The file should contain the following command:

```
java -classpath c:\xml4j\xml4j.jar;c:\xml4j\xml4jsamples.jar
➔XJParse -p com.ibm.xml.parsers.ValidatingSAXParser %1
```

Now you can validate any XML file with the following (shorter) command:

```
validate abook.xml
```

Entities and Notations

As already mentioned in the previous chapter, XML doesn't work with files but with entities. Entities are the physical representation of XML documents. Although entities usually are stored as files, they need not be.

In XML the document, its DTD, and the various files it references (images, stock-phrases, and so on) are entities. The document itself is a special entity because it is the starting point for the XML processor. The entity of the document is known as the *document entity*.

XML does not dictate how to store and access entities. This is the task of the XML processor and it is system specific. The XML processor might have to download entities or it might use a local catalog file to retrieve the entities.

In Chapter 7, "The Parser and DOM," you'll see how SAX parsers (a SAX parser is one example of an XML processor) enable the application to retrieve entities from databases or other sources.

XML has many types of entities, classified according to three criteria: *general* or *parameter entities*, *internal* or *external entities*, and *parsed* or *unparsed entities*.



EXAMPLE

General and Parameter Entities

General entity references can appear anywhere in text or markup. In practice, general entities are often used as macros, or shorthand for a piece of text. External general entities can reference images, sound, and other documents in non-XML format. Listing 3.10 shows how to use a general entity to replace some text.

Listing 3.10: General Entity

```
<?xml version="1.0"?>
<!DOCTYPE address-book [
<!ENTITY jacksmith
'<entry>
  <name><fname>Jack</fname><lname>Smith</lname></name>
  <tel>513-555-3465</tel>
  <email href="mailto:jsmith@emailaholic.com"/>
</entry>'>
]>
<address-book>
  &jacksmith;
</address-book>
```

General entities are declared with the markup `<!ENTITY` followed by the entity name, the entity definition, and the customary right angle bracket.

TIP

General entities also are often used to associate a mnemonic with character references as in

```
<!ENTITY icirc "&#238;">
```

As we saw in Chapter 2, “The XML Syntax,” the following entities are pre-defined in XML: “<”, “&”, “>”, “'”, and “"”.

Parameter entity references can only appear in the DTD. There is an extra % character in the declaration before the entity name. Parameter entity references also replace the ampersand with a percent sign as in

```
<!ENTITY % boolean "(true | false) 'false'">
<!ELEMENT tel (#PCDATA)>
<!ATTLIST tel preferred %boolean;>
```


Parameter entities have many applications. You will learn how to use parameter entities in the following sections: “Internal and External Entities,” “Conditional Sections,” “Designing DTDs from an Object Model.”

CAUTION

The previous example is valid only in the external subset of a DTD. In the internal subset, parameter entities can appear only where markup declaration can appear.

Internal and External Entities

XML also distinguishes between internal and external entities. Internal entities are stored in the document, whereas external entities point to a system or public identifier. Entity identifiers are identical to DTD identifiers (in fact, the DTD is a special entity).

The entities in the previous sections were internal entities because their value was declared in the entity definition. External entities, on the other hand, reference content that is not part of the current document.

TIP

External entities might start with an XML declaration—for example, to declare a special encoding.

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

External general entities can be parsed or unparsed. If parsed, the entity must contain valid XML text and markup. External parsed entities are used to share text across several documents, as illustrated by Listing 3.11.

**EXAMPLE**

In Listing 3.11, the various entries are stored in separate entities (separate files). The address book combines them in a document.

Listing 3.11: Using External Entities

```
<?xml version="1.0"?>
<!DOCTYPE address-book [
  <!ENTITY johndoe SYSTEM "johndoe.ent">
  <!ENTITY jacksmith SYSTEM "jacksmith.ent">
]>
<address-book>
  &johndoe;
  &jacksmith;
</address-book>
```

Where the file “johndoe.ent” contains:

```
<entry>
  <name>John Doe</name>
```

```

<address>
  <street>34 Fountain Square Plaza</street>
  <region>OH</region>
  <postal-code>45202</postal-code>
  <locality>Cincinnati</locality>
  <country>US</country>
</address>
</entry>

```

And “jacksmith.ent” contains

```

<entry>
  <name><fname>Jack</fname><lname>Smith</lname></name>
  <tel>513-555-3465</tel>
  <email href="mailto:jsmith@emailaholic.com"/>
</entry>

```



EXAMPLE

However, unparsed entities are probably the most helpful external general entities. Unparsed entities are used for non-XML content, such as images, sound, movies, and so on. Unparsed entities provide a mechanism to load non-XML data into a document.

The XML processor treats the unparsed entity as an opaque block, of course. By definition, it does not attempt to recognize markup in unparsed entities.

A notation must be associated with unparsed entities. Notations are explained in more detail in the next section but, in a nutshell, they identify the type of a document, such as GIF, JPEG, or Windows bitmap for images. The notation is introduced by the NDATA keyword:

```

<!ENTITY logo SYSTEM "http://catwoman.pineapplesoft.com/logo.gif"
      NDATA GIF>

```



EXAMPLE

External parameter entities are similar to external general entities. However, because parameter entities appear in the DTD, they must contain valid XML markup.

External parameter entities are often used to insert the content of a file in the markup. Let’s suppose we have created a list of general entities for every country, as in Listing 3.12 (saved in the file `countries.ent`).

Listing 3.12: A List of Entities for the Countries

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!ENTITY be "Belgium">

```

```

<!ENTITY ch "Switzerland">
<!ENTITY de "Germany">
<!ENTITY it "Italy">
<!ENTITY jp "Japan">
<!ENTITY uk "United Kingdom">
<!ENTITY us "United States">
<!-- and more -->

```

**EXAMPLE**

Creating such a list is a large effort. We would like to reuse it in all our documents. The construct illustrated in Listing 3.13 pulls the list of countries from `countries.ent` in the current document. It declares a parameter entity as an external entity and it immediately references the parameter entity. This effectively includes the external list of entities in the DTD of the current document.

Listing 3.13: Using External Parameter Entities

```

<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd" [
<!ENTITY % countries SYSTEM "countries.ent">
%countries;
]>
<address>
  <street>34 Fountain Square Plaza</street>
  <region>Ohio</region>
  <postal-code>45202</postal-code>
  <locality>Cincinnati</locality>
  <country>&us;</country>
</address>

```

CAUTION

Given the limitation on parameter entities in the internal subset of the DTD, this is the only sensible application of parameter entities in the internal subset.

Notation

Because the XML processor cannot process unparsed entities, it needs a mechanism to associate them with the proper tool. In the case of an image, it could be an image viewer.

**EXAMPLE**

Notation is simply a mechanism to declare the type of unparsed entities and associate them, through an identifier, with an application.

```

<!NOTATION GIF89a PUBLIC "-//CompuServe//NOTATION Graphics
  Interchange Format 89a//EN" "c:\windows\kodakprv.exe">

```

**EXAMPLE**

This declaration is unsafe because it points to a specific application. The application might not be available on another computer or it might be available but from another path. If your system has defined the appropriate file associations, you can get away with a declaration such as

```
<!NOTATION GIF89a SYSTEM "GIF">
<!NOTATION GIF89a SYSTEM "image/gif">
```

The first notation uses the filename, while the second uses the MIME type.

Managing Documents with Entities

External entities are helpful to modularize and help manage large DTDs and large document sets.

The idea is very simple: Try to divide your work into smaller pieces that are more manageable. Save each piece in a separate file and include them in your document with external entities.

Also try to identify pieces that you can reuse across several applications. It might be a list of entities (such as the list of countries) or a list of notations, or some text (such as a copyright notice that must appear on every document). Place them in separate files and include them in your documents through external entities.

Figure 3.3 shows how it works. Notice that some files are shared across several documents.

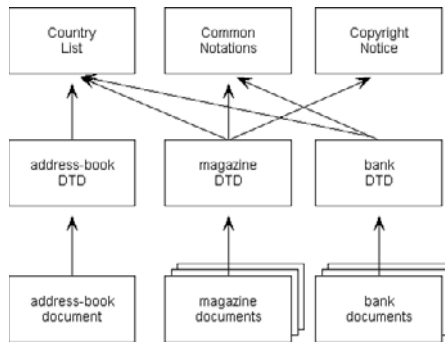


Figure 3.3: Using external entities to manage large projects

This is like eating a tough steak: You have to cut the meat into smaller pieces until you can chew it.

Conditional Sections

As your DTDs mature, you might have to change them in ways that are partly incompatible with previous usage. During the migration period, when you have new and old documents, it is difficult to maintain the DTD.

To help you manage migrations and other special cases, XML provides *conditional sections*. Conditional sections are included or excluded from the DTD depending on the value of a keyword. Therefore, you can include or exclude a large part of a DTD by simply changing one keyword.



EXAMPLE

Listing 3.13 shows how to use conditional sections. The strict parameter entity resolves to INCLUDE. The lenient parameter entity resolves to IGNORE. The application will use the definition of name in the %strict; section ((fname, lname)) and ignores the definition in the %lenient; section ((#PCDATA | fname | lname)*).

Listing 3.13: Using Conditional Sections

```
<!ENTITY % strict 'INCLUDE'>
<!ENTITY % lenient 'IGNORE'>

<![%strict;[
<!-- a name is a first name and a last name -->
<!ELEMENT name (fname, lname)>
]]>
<![%lenient;[
<!-- name is made of string, first name
and last name. This is a very flexible
model to accommodate exotic name -->
<!ELEMENT name (#PCDATA | fname | lname)*>
]]>
```

However, to revert to the lenient definition of name, it suffices to invert the parameter entity declaration:

```
<!ENTITY % strict 'IGNORE'>
<!ENTITY % lenient 'INCLUDE'>
```

Designing DTDs

Now that you understand what DTDs are for and that you understand how to use them, it is time to look at how to create DTDs. DTD design is a creative and rewarding activity.

It is not possible, in this section, to cover every aspect of DTD design. Books have been devoted to that topic. Use this section as guidance and remember that practice makes proficient.

Yet, I would like to open this section with a plea to use existing DTDs when possible. Next, I will move into two examples of the practical design of practically designing DTDs.

Main Advantages of Using Existing DTDs

There are many XML DTDs available already and it seems more are being made available every day. With so many DTDs, you might wonder whether it's worth designing your own.

I would argue that, as much as possible, you should try to reuse existing DTDs. Reusing DTDs results in multiple savings. Not only do you not have to spend time designing the DTD, but also you don't have to maintain and update it.

However, designing an XML application is not limited to designing a DTD. As you will learn in Chapter 5, "XSL Transformation," and subsequent chapters, you might also have to design style sheets, customize tools such as editors, and/or write special code using a parser.

This adds up to a lot of work. And it follows the "uh, oh" rule of project planning: Uh, oh, it takes more work than I thought." If at all possible, it pays to reuse somebody else's DTD.

The first step in a new XML project should be to search the Internet for similar applications. I suggest you start at www.oasis-open.org/sgml/xml.html. The site, maintained by Robin Cover, is the most comprehensive list of XML links.

In practice, you are likely to find DTDs that almost fit your needs but aren't exactly what you are looking for. It's not a problem because XML is extensible so it is easy to take the DTD developed by somebody else and adapt it to your needs.

Designing DTDs from an Object Model

I will take two examples of DTD design. In the first example, I will start from an object model. This is the easiest solution because you can reuse the objects defined in the model. In the second example, I will create a DTD from scratch.



EXAMPLE

Increasingly, object models are made available in UML. UML is the Unified Modeling Language (yes, there is an ML something that does not stand for markup language). UML is typically used for object-oriented applications such as Java or C++ but the same models can be used with XML.

An object model is often available when XML-enabling an existing Java or C++ application. Figure 3.4 is a (simplified) object model for bank accounts. It identifies the following objects:

- “Account” is an abstract class. It defines two properties: the balance and a list of transactions.
- “Savings” is a specialized “Account” that represents a savings account; interest is an additional property.
- “Checking” is a specialized “Account” that represents a checking account; rate is an additional property.
- “Owner” is the account owner. An “Account” can have more than one “Owner” and an “Owner” can own more than one “Account.”

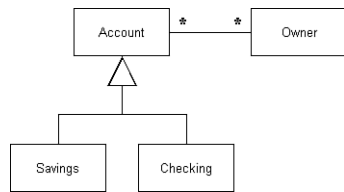


Figure 3.4: *The object model*

The application we are interested in is Web banking. A visitor would like to retrieve information about his or her various bank accounts (mainly his or her balance).

The first step to design the DTD is to decide on the root-element. The top-level element determines how easily we can navigate the document and access the information we are interested in. In the model, there are two potential top-level elements: Owner or Account.

Given we are doing a Web banking application, Owner is the logical choice as a top element. The customer wants his list of accounts.

Note that the choice of a top-level element depends heavily on the application. If the application were a financial application, examining accounts, it would have been more sensible to use account as the top-level element.

At this stage, it is time to draw a tree of the DTD under development. You can use a paper, a flipchart, a whiteboard, or whatever works for you (I prefer flipcharts).

In drawing the tree, I simply create an element for every object in the model. Element nesting is used to model object relationship.

Figure 3.5 is a first shot at converting the model into a tree. Every object in the original model is now an element. However, as it turns out, this tree is both incorrect and suboptimal.



Figure 3.5: A first tree for the object model

Upon closer examination, the tree in Figure 3.5 is incorrect because, in the object model, an account can have more than one owner. I simply cannot add the owner element into the account because this would lead to infinite recursion where an account includes its owner, which itself includes the account, which includes the owner, which... You get the picture.

The solution is to create a new element co-owner. To avoid confusion, I decided to rename the top-level element from owner to accounts. The new tree is in Figure 3.6.

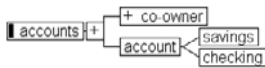


Figure 3.6: The corrected tree

The solution in Figure 3.6 is a correct implementation of the object model. To evaluate how good it is, I like to create a few sample documents that follow the same structure. Listing 3.14 is a sample document I created.

Listing 3.14: Sample Document

```

<?xml version="1.0"?>
<accounts>
  <co-owner>John Doe</co-owner>
  <co-owner>Jack Smith</co-owner>
  <account>
    <checking>170.00</checking>
  </account>
  <co-owner>John Doe</co-owner>
  <account>
    <savings>5000.00</savings>
  </account>
</accounts>
  
```

This works but it is inefficient. The checking and savings elements are completely redundant with the account element. It is more efficient to treat

account as a parameter entity that groups the commonality between the various accounts. Figure 3.7 shows the result. In this case, the parameter entity is used to represent a type.

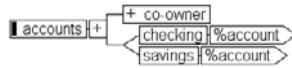


Figure 3.7: *The tree, almost final*

We're almost there. Now we need to flesh out the tree by adding the object properties. I chose to create new elements for every property (see the following section "On Elements Versus Attributes").

Figure 3.8 is the final result. Listing 3.15 is a document that follows the structure. Again, it's useful to write a few sample documents to check whether the DTD makes sense. I can find no problems with this structure in Listing 3.15.

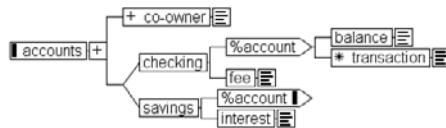


Figure 3.8: *The final tree*

Listing 3.15: A Sample Document

```
<?xml version="1.0"?>
<accounts>
  <co-owner>John Doe</co-owner>
  <co-owner>Jack Smith</co-owner>
  <checking>
    <balance>170.00</balance>
    <transaction>-100.00</transaction>
    <transaction>-500.00</transaction>
    <fee>4.00</fee>
  </checking>
  <co-owner>John Doe</co-owner>
  <savings>
    <balance>5000.00</balance>
    <interest>212.50</interest>
  </savings>
</accounts>
```

Having drawn the tree, it is trivial to turn it into a DTD. It suffices to list every element in the tree and declare their content model based on their children. The final DTD is in Listing 3.16.

Listing 3.16: The DTD for Banking

```
<!ENTITY % account      "(balance,transaction*)">
<!ELEMENT accounts      (co-owner+, (checking | savings))+>
<!ELEMENT co-owner      (#PCDATA)>
<!ELEMENT checking      (%account;, fee)>
<!ELEMENT savings      (%account;, interest)>
<!ELEMENT fee           (#PCDATA)>
<!ELEMENT interest      (#PCDATA)>
<!ELEMENT balance       (#PCDATA)>
<!ELEMENT transaction   (#PCDATA)>
```

Now I have to publish this DTD under a URI. I like to place versioning information in the URI (version 1.0, and so on) because if there is a new version of the DTD, it gets a different URI with the new version. It means the two DTDs can coexist without problem.

It also means that the application can retrieve the URI to know which version is in use.

```
http://catwoman.pineapplesoft.com/dtd/accounts/1.0/accounts.dtd
```

If I ever update the DTD (it's a very simplified model so I can think of many missing elements), I'll create a different URI with a different version number:

```
http://catwoman.pineapplesoft.com/dtd/accounts/2.0/accounts.dtd
```

You can see how easy it is to create an XML DTD from an object model. This is because XML tree-based structure is a natural mapping for objects.

As more XML applications will be based on object-oriented technologies and will have to integrate with object-oriented systems written in Java, CORBA, or C++, I expect that modeling tools will eventually create DTDs automatically.

Already modeling tools such as Rational Rose or Together/J can create Java classes automatically. Creating DTDs seems like a logical next step.

On Elements Versus Attributes

As you have seen, there are many choices to make when designing a DTD. Choices include deciding what will become of an element, a parameter entity, an attribute, and so on.

Deciding what should be an element and what should be an attribute is a hot debate in the XML community. We will revisit this topic in Chapter 10, “Modeling for Flexibility,” but here are some guidelines:

- The main argument in favor of using attributes is that the DTD offers more controls over the type of attributes; consequently, some people argue that object properties should be mapped to attributes.
- The main argument for elements is that it is easier to edit and view them in a document. XML editors and browsers in general have more intuitive handling of elements than of attributes.

I try to be pragmatic. In most cases, I use element for “major” properties of an object. What I define as major is all the properties that you manipulate regularly.

I reserve attributes for ancillary properties or properties that are related to a major property. For example, I might include a currency indicator as an attribute to the balance.

Creating the DTD from Scratch

Creating a DTD without having the benefit of an object model results in more work. The object model provides you with ready-made objects that you just have to convert in XML. It also has identified the properties of the objects and the relationships between objects.

However, if you create a DTD from scratch, you have to do that analysis as well.

A variant is to modify an existing DTD. Typically, the underlying DTD does not support all your content (you need to add new elements/attributes) or is too complex for your application (you need to remove elements/attributes).

This is somewhat similar to designing a DTD from scratch in the sense that you will have to create sample documents and analyze them to understand how to adapt the proposed DTD.

On Flexibility

When designing your own DTD, you want to prepare for evolution. We’ll revisit this topic in Chapter 10 but it is important that you build a model that is flexible enough to accommodate extensions as new content becomes available.

The worst case is to develop a DTD, create a few hundred or a few thousand documents, and suddenly realize that you are missing a key piece of information but that you can’t change your DTD to accommodate it. It’s bad because it means you have to convert your existing documents.

To avoid that trap you want to provide as much structural information as possible but not too much. The difficulty, of course, is in striking the right balance between enough structural information and too much structural information.

You want to provide enough structural information because it is very easy to degrade information but difficult to clean degraded information.

Compare it with a clean, neatly sorted stack of cards on your desk. It takes half a minute to knock it down and shuffle it. Yet it will take the best part of one day to sort the cards again.

The same is true with electronic documents. It is easy to lose structural information when you create the document. And if you lose structural information, it will be very difficult to retrieve it later on.

**EXAMPLE**

Consider Listing 3.17, which is the address book in XML. The information is highly structured—the address is broken down into smaller components: street, region, and so on.

Listing 3.17: An Address Book in XML

```
<?xml version="1.0"?>
<!DOCTYPE address-book SYSTEM "address-book.dtd">
<!-- loosely inspired by vCard 3.0 -->
<address-book>
  <entry>
    <name>John Doe</name>
    <address>
      <street>34 Fountain Square Plaza</street>
      <region>OH</region>
      <postal-code>45202</postal-code>
      <locality>Cincinnati</locality>
      <country>US</country>
    </address>
    <tel preferred="true">513-555-8889</tel>
    <tel>513-555-7098</tel>
    <email href="mailto:jdoe@emailaholic.com"/>
  </entry>
  <entry>
    <name><fname>Jack</fname><lname>Smith</lname></name>
    <tel>513-555-3465</tel>
    <email href="mailto:jsmith@emailaholic.com"/>
  </entry>
</address-book>
```

Listing 3.18 is the same information as text. The structure is lost and, unfortunately, it will be difficult to restore the structure automatically. The software would have to be quite intelligent to go through Listing 3.18 and retrieve the entry boundaries as well as break the address in its components.

Listing 3.18: The Address Book in Plain Text

```
John Doe
34 Fountain Square Plaza
Cincinnati, OH 45202
US
513-555-8889 (preferred)
513-555-7098
jdoe@emailaholic.com
Jack Smith
513-555-3465
jsmith@emailaholic.com
```

However, as you design your structure, be careful that it remains usable. Structures that are too complex or too strict will actually lower the quality of your document because it encourages users to cheat.

Consider how many electronic commerce Web sites want a region, province, county, or state in the buyer address. Yet many countries don't have the notion of region, province, county, or state or, at least, don't use it for their addresses.

Forcing people to enter information they don't have is asking them to cheat. Keep in mind the number one rule of modeling: Changes will come from the unexpected. Chances are that, if your application is successful, people will want to include data you had never even considered. How often did I include for "future extensions" that were never used? Yet users came and asked for totally unexpected extensions.

There is no silver bullet in modeling. There is no foolproof solution to strike the right balance between extensibility, flexibility, and usability. As you grow more experienced with XML and DTDs, you also will improve your modeling skills.

My solution is to define a DTD that is large enough for all the content required by my application but not larger. Still, I leave hooks in the DTD—places where it would be easy to add a new element, if required.

**EXAMPLE**

Modeling an XML Document

The first step in modeling XML documents is to create documents. Because we are modeling an address book, I took a number of business cards and created documents with them. You can see some of the documents I created in Listing 3.20.

Listing 3.20: Examples of XML Documents

```
<address-book>
  <entry>
    <name><fname>John</fname><lname>Doe</lname></name>
    <address>
      <street>34 Fountain Square Plaza</street>
      <state>OH</state>
      <zip>45202</zip>
      <locality>Cincinnati</locality>
      <country>US</country>
    </address>
    <tel>513-555-8889</tel>
    <email href="mailto:jdoe@emailaholic.com"/>
  </entry>
  <entry>
    <name><fname>Jean</fname><lname>Dupont</lname></name>
    <address>
      <street>Rue du Lombard 345</street>
      <postal-code>5000</postal-code>
      <locality>Namur</locality>
      <country>Belgium</country>
    </address>
    <email href="mailto:jdupont@emailaholic.com"/>
  </entry>
  <entry>
    <name><fname>Olivier</fname><lname>Rame</lname></name>
    <email href="mailto:orame@emailaholic.com"/>
  </entry>
</address-book>
```

As you can see, I decided early on to break the address into smaller components. In making these documents, I tried to reuse elements over and over again. Very early in the project, it was clear there would be a name element, an address element, and more.

Also, I decided that addresses, phone numbers, and so on would be conditional. I have incomplete entries in my address book and the XML version must be able to handle it as well.

I looked at commonalities and I found I could group postal code and zip code under one element. Although they have different names, they are the same concepts.

This is the creative part of modeling when you list all possible elements, group them, and reorganize them until you achieve something that makes sense. Gradually, a structure appears.

Building the DTD from this example is easy. I first draw a tree with all the elements introduced in the document so far, as well as their relationship. It is clear that some elements such as state are optional. Figure 3.9 shows the tree.

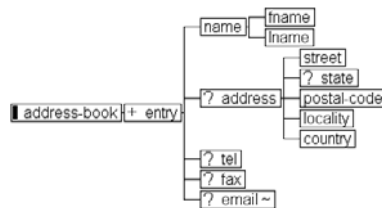


Figure 3.9: *The updated tree*

This was fast to develop because the underlying model is simple and well known. For a more complex application, you would want to spend more time drafting documents and trees.

At this stage, it is a good idea to compare my work with other similar works. In this case, I choose to compare with the vCard standard (RFC 2426). vCard (now in its third version) is a standard for electronic business cards.

vCard is a very extensive standard that lists all the fields required in an electronic business card. vCard, however, is too complicated for my needs so I don't want to simply duplicate that work.

By comparing the vCard structure with my structure, I realized that names are not always easily broken into first and last names, particularly foreign names. I therefore provided a more flexible content model for names.

I also realized that address, phone, fax number, and email address might repeat. Indeed, it didn't show up in my sample of business cards but there are people with several phone numbers or email addresses. I introduced a repetition for these as well as an attribute to mark the preferred address. The attribute has a default value of false.

In the process, I picked the name “region” for the state element. For some reason, I find region more appealing.

Comparing my model with vCard gave me the confidence that the simple address book can cope with most addresses used. Figure 3.10 is the result.

TIP

There is a group working on the XML-ization of the vCard standard. Its approach is different: It starts with vCard as its model, whereas this example starts from an existing document and uses vCard as a check.

Yet, it is interesting to compare the XML version of vCard (available from www.imc.org/ietf-vcard-xml) with the DTD in this chapter. It proves that there is more than one way to skin a cat.

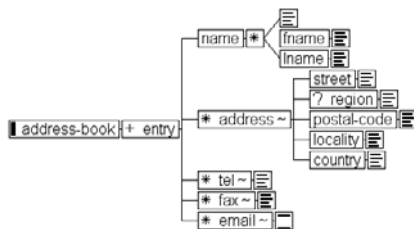


Figure 3.10: The final tree

Again converting the tree in a DTD is trivial. Listing 3.21 shows the result.

Listing 3.21: A DTD for the Address Book

```
<!ENTITY % boolean "(true | false) 'false'">

<!-- top-level element, the address book
      is a list of entries                                -->
<!ELEMENT address-book (entry+)>

<!-- an entry is a name followed by
      addresses, phone numbers, etc.                    -->
<!ELEMENT entry (name,address*,tel*,fax*,email*)>

<!-- name is made of string, first name
      and last name. This is a very flexible
      model to accommodate exotic name                  -->
<!ELEMENT name (#PCDATA | fname | lname)*>
<!ELEMENT fname (#PCDATA)>
```



```

<!ELEMENT lname    (#PCDATA)>

<!-- definition of the address structure
      if several addresses, the preferred
           attribute signals the "default" one  -->
<!ELEMENT address  (street,region?,postal-code,locality,country)>
<!ATTLIST address  preferred (true | false)  "false">
<!ELEMENT street   (#PCDATA)>
<!ELEMENT region   (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
<!ELEMENT locality (#PCDATA)>
<!ELEMENT country  (#PCDATA)>

<!-- phone, fax and email, same preferred
           attribute as address  -->
<!ELEMENT tel      (#PCDATA)>
<!ATTLIST tel      preferred (true | false)  "false">
<!ELEMENT fax      (#PCDATA)>
<!ATTLIST fax      preferred (true | false)  "false">
<!ELEMENT email    EMPTY>
<!ATTLIST email    href CDATA                #REQUIRED
                   preferred (true | false)  "false">

```

Naming of Elements

Again, modeling requires imagination. One needs to be imaginative and keep an open mind during the process. Modeling also implies making decisions on the name of elements and attributes.

As you can see, I like to use meaningful names. Others prefer to use meaningless names or acronyms. Again, as is so frequent in modeling, there are two schools of thought and both have very convincing arguments. Use what works better for you but try to be consistent.

In general, meaningful names

- are easier to debug
- provide some level of document for the DTD.

However, a case can be made for acronyms:

- Acronyms are shorter, and therefore more efficient.
- Acronyms are less language-dependent.

- Name choice should not be a substitute for proper documentation; meaningless tags and acronyms might encourage you to properly document the application.

A Tool to Help

I find drawing trees on a piece of paper an exercise in frustration. No matter how careful you are, after a few rounds of editing, the paper is unreadable and modeling often requires several rounds of editing!

Fortunately, there are very good tools on the market to assist you while you write DTDs. The trees in this book were produced by Near & Far from Microstar (www.microstar.com).

Near & Far is as intuitive as a piece of paper but, even after 1,000 changes, the tree still looks good. Furthermore, to convert the tree in a DTD, it suffices to save it. No need to remember the syntax, which is another big plus.

Figure 3.11 is a screenshot of Near & Far.



EXAMPLE

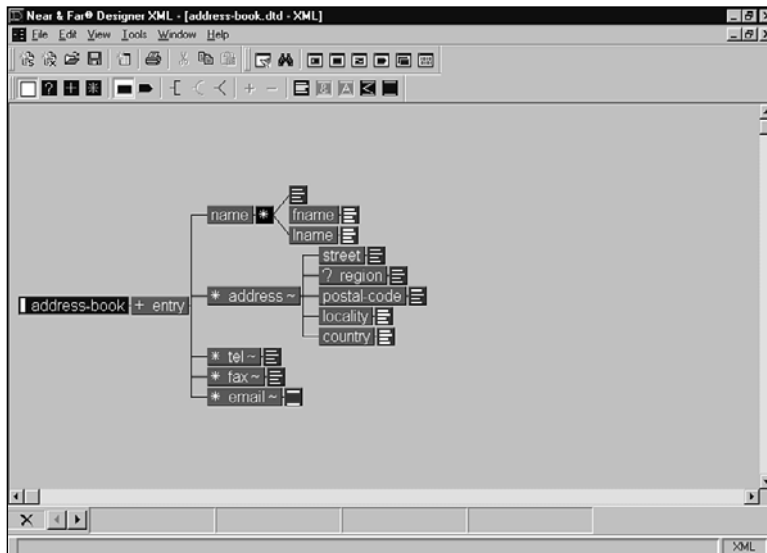


Figure 3.11: Using a modeling tool

New XML Schemas

The venerable DTD is very helpful. It provides valuable services to the application developer and the XML author. However, DTD originated in publishing and it shows.

For one thing, content is limited to textual content. Also, it is difficult to put in repetition constraints: You cannot say that an element can appear only four times. It's 0, 1, or infinite.

Furthermore, the DTD is based on a special syntax that is different from the syntax for XML documents. It means that it is not possible to use XML tools, such as editors or browsers, to process DTD!

These so-called limitations of the DTD are inherited directly from SGML. XML was originally designed as a subset of SGML and therefore it could not differ too much from the SGML DTD.

However, as XML takes a life of its own, people would like a new, more modern, replacement for the DTD. Various groups have made several proposals. Collectively, these proposals are known as schemas. The details of the proposals vary greatly but all:

- propose to use the same syntax as XML documents
- improve XML data typing to support not only strings but also numbers, dates, and so on
- introduce object-oriented concepts such as inheritance (an element could inherit from another)

The W3C has formed a working group to develop a new standard based on the existing proposals. At the time of this writing, the effort has just started and little is known about the final result.

You can find up-to-date information on new XML schemas on the W3C Web site at www.w3.org/XML.

The main proposals being considered are

- XML-Data, which offers types inspired from SQL types.
- DCD (Document Content Description), positioned as a simplified version of XML-Data.
- SOX (Schema for Object-oriented XML), as the name implies, is heavy on object-orientation aspects.
- DDML (Document Definition Markup Language), developed by the XML-Dev mailing list. It is intended as a simple solution to form a basis for future work.

What's Next

The next chapter is dedicated to the namespace proposal. Namespace is an often-overlooked but very useful standard that greatly enhances XML extensibility.

Symbols

- * (asterisk), 73
- , (comma), 73-74
- + (plus), 73
- ? (question mark), 73
- _ (underscore), 45
- | (vertical bar), 73-74

A

AAP (Association of American Publishers), 20

accessing

- classpaths, 242
- entities, 85
- Web sites, 6

acronyms, 103

Active Server Page (ASP), 376

ActiveX, 294

adding

- CDATA sections, 55
- comments, 50
- elements, 128
- headers, 276
- links (XLink), 326
- nodes, 274-276
- non-XML statements, 53-54
- objects, 275
- properties, 95
- tiers, 348

address books, 308-309, 312

Ælfred, 237, 460

alignment of text, 177

alternate style sheet, 310-311

ancestors, Java inheritance, 477

anchor elements, 140, 324

anonymous boxes, 171

APIs, *see* DOM

appendChild() method, 277, 283

appendData() method, 278

applications, 231

business documents, 63-65

client/server, 346-347

data, 29-32

documents, 29

event handlers, 233

n-tiered, 345, 348

object-based interface, 194, 196

publishing, 62-63

scientific, 7

server-side, 458-459

standards, 197-198

three-tiered, 347-348

XML architecture, 193-194

applying style sheets, 145-148

architecture

applications, 193-194

classes, 260-261

parsers, 193-194

archives, JAR (Java), 481-482

arguments
 arrays declaring, 465
 XSLT functions, 137

ASP, 6, 295-296

Association of American Publishers (AAP), 20

asterisk (*), 73

attaching, *see* linking

Attr object, 217-218

attributes, 46-47, 97
 Attr object, 217-218
 content, 339-340
 conversion utility, 212, 214-216
 creating, 278
 currency, 211, 217
 data, 218
 DTDs, 96
 elements, 75
 enumerated type, 75
 exchange rates, 211, 216-217
 HTML, 21-23
 languages, 53
 links (XLink), 324
 NamedNodeMap object, 217
 names, 217
 nodes, 202
 parents, 340
 price list, 216-217
 properties, 342
 SAX, 244-245
 selectors, 181
 signatures, 328
 spaces, 53
 string, 75
 structure, 218
 tokenized, 75
 values, 60, 75-76, 217, 341
 XSLT, 136, 305

B

backend parsers, 193

background, 178-179

batch files, validation, 85

block boxes, 171

boolean primitive type, 465

borders
 boxes, 175
 color, 178

boxes
 borders, 175
 display property, 174
 flow objects, 169-172
 margins, 174-175
 padding, 175

browsers
 compatibility, 9
 conversion utility, 225-227
 CSS, 163, 227-228
 documents, 295-296
 DOM, 199, 225-229
 DTDs, 296
 elements, 341
 frames, 225
 images, 324
 InDelv, 187-189
 parsers, 219
 style sheets, 145
 XLink, 327
 XML, 36-37, 129-130

built-in
 business documents semantics, 64-65
 XSLT templates, 138-139

business documents application, 63-65

C

C++, 48

calling
 templates, 154
 XSLT, 303-304

CALS (Continuous Acquisition and Life-cycle Support) standard MIL-M-28001B, 20

canvases, 168

Cascading Style Sheets, *see* CSS

case-sensitive
 element names, 45
 syntax, 60

CGI (Common Gateway Interface), 6, 376, 473

channels
 defining, 65-66
 subscriptions, 65-66

character data (CDATA), 43, 54-55
 methods, 278
 references, 53
 Unicode, 51
 whitespace() event, 244

checkout (XCommerce), 351, 407-417

child elements, 48
 nodes, 202, 275
 selectors (CSS2), 180
 trees, 206

Clark, James, 460

classes
 architecture, 260-261
 imports, 459, 466, 468
 package, 469
 private, 469
 protected, 469
 public, 469
 SAX, 242
 static modifier, 467

classpaths
 accessing, 242
 Java, 475, 480-481
 libraries, 381-382, 384
 XCommerce, 381-382

client/server applications, 346-347
 4GL, 346
 common formats, 357-359

- cost, 347
- database servers, 347
- distributed, 346
- email, 346-347
- file servers, 346
- middle tier, 372-375
- middleware, 356-357
- print servers, 346
- roles of computers, 346
- services, 346
- see also* browsers; server-side applications
- cloneNode() method, 277**
- code, *see* source code**
- colon, 45**
- color, 173, 178**
- combining style sheets, 319**
- comma (,), 73-74**
- command-line version (Java), 84**
- comments, 166**
 - adding, 50
 - Java, 469-470
- commerce applications, *see* XCommerce**
- common elements, 319-320**
- common formats, 357-359**
- Common Gateway Interface (CGI), 6, 376, 473**
- Common Object Request Broker Architecture, *see* middleware**
- companion standards, 32**
- comparing**
 - CSS with XSLT, 162-163
 - Java with JavaScript, 223-224
 - prices, 312-313
- ComparisonMachine class, 260**
- compatibility**
 - browsers, 9
 - CSS, 164
 - XSLT, 128
- compiling**
 - data extracts, 157
 - Java files, 383-384
 - parsers, 193
 - SAX, 241-242
- components, server-side applications, 458-459**
- computer platforms, 219**
- conditional documents/sections, 91, 101**
- configuration files (XCommerce), 382-383**
 - attributes, 278
 - document handlers, 243
 - entity resolvers, 243
 - named items, 217
 - request headers, 294
- conflicts in namespaces, 108, 112**
- connectors**
 - DTD, 73-74
 - or, 338
- constructors (Java), 467**
- content**
 - attributes, 339-340
 - creating, 182
 - DTDs, 74
 - elements, 339-340
 - model, 71
- Continuous Acquisition and Life-cycle Support (CALs), 20**
- control (Java), 468-469**
- conversion**
 - attributes, 212, 214-216
 - browsers, 225-227
 - currency, 199-200, 271
 - DTDs, 128, 322
 - formats, 29
 - HTML, 332
 - Java, 221-223
 - JavaScript, 200-201
 - text, 141-144
 - XSLT style sheets, 131-132
- CORBA objects, 221**
- core APIs (Java), 482**
- cost, 359**
 - client/server applications, 347
 - domain name registration, 116
 - middle tier, 366
 - Notepad, 61
- creating**
 - attributes, 277-278
 - CDATA section, 277
 - comments, 277
 - content, 182
 - document fragments, 187, 277
 - DTDs, 97-99
 - elements, 277
 - entity references, 277
 - HTML, 284
 - DOM*, 279-281, 283
 - JavaScript*, 280-281, 284-288
 - non-XML data structures*, 291-292
 - Java, 466
 - nodes, 281-282
 - objects, 140
 - parsers, 296
 - processing instructions, 277
 - style sheets, 149-151
 - text nodes, 277
 - top-level elements, 138, 282-283
 - URLs, 114, 117

CSS (Cascading Style Sheets)

advanced style sheets, 179-180
 boxes, 174-175
 browsers, 163, 227-228
 color, 173, 178
 comments, 166
 compatibility, 164
 development, 163
 documents, 126, 165-166, 187
 editors, 182-184
 flow objects, 168-172, 189
 fonts, 176-178
 images, 178-179
 priorities, 167-168
 properties, 168, 172-173
 selectors, 166-167
 simple, 164-165
 tables of contents, 185-186
 text, 177
 versions, 163
 XSLFO, 187-189
 XSLT, 162-163, 185

CSS2

attribute selectors, 181
 child selectors, 180
 content, 182
 development, 163
 sibling selectors, 181
 style sheets, 182

currency, 199-200, 211, 217, 271**customizing middle tier, 367****D****data**

applications, 29-32
 attributes, 218
 directories, 383
 EDI, 322

extracting, 155-157
 HTML, 284
 non-DOM, 288-289
 styling, 157-158
 tier, 348, 353-355, 429-444

databases

DOM, 229
 offloading, 7
 relational, 30
 reloading, 7
 servers, 347
 standards, 197
 updating, 32
 XCommerce, 361-364, 384-385

DCD (Document Content Description), 105**DCOM, see middleware****DDML (Document Definition Markup Language), 105****declarations**

arrays, 465
 attributes, 75, 80
 document types, 76-77
 elements, 71-72, 77
 encoding, 51
 entities, 80, 246
 HTML, 153
 namespaces, 114, 120
 notations, 246
 SAX events, 243-244
 XML, 49

default settings

attribute values, 76, 80, 84
 DTDs, 303-304

defining

channels, 65-66
 elements, 80, 162
 entities, 52

deleteData() method, 278**descendants**

Java inheritance, 476-477
 Text object, 206

designing DTDs

attributes, 96-97
 automating, 96
 elements, 96-97
 nested elements, 93
 object models, 92-93
 owner elements, 94
 parameter entities, 95
 properties, 95
 root-elements, 93-94
 top-level elements, 93
 trees, 95
 URIs, 96

Web sites, 26-27

destructive document transformations, 332**deterministic content model, 74****development of namespaces, 120****digital signatures, 328-329****directories**

Java, 480-481
 XCommerce, 383

display property, 174**DocBook, 20****docs directory, 383****documents**

applications, 29
 attributes, 278
 business, 63-65
 Content Description (DCD), 105
 creating, 187
DOM, 279-281, 283
HTML, 284
JavaScript, 280-281, 284-288
non-XML data structures, 291-292

CSS, 165-166

- declarations, 76-77
- Document Definition Markup Language (DDML), 105
- DTDs, 79-80
- Dump service, 293
- elements, 265-266, 335-336
- ending, 244
- entities, 52, 85-89
- events, 242-243
- format conversion, 29
- JavaScript, 295
- markups, 334-336
- methods, 277
- modeling, 100-104
- nodes, 274-276, 281-282
- objects, 203-204
- posting to Web servers, 293-294
- retrieving from Web servers, 293-295
- saving, 295-296
- semantics, 58, 64-65
- sending to the server, 292-294
- sharing, 321
- starting, 244
- storage, 229
- structural information, 10-14, 18, 63, 266, 334
- top-level elements, 282-283
- transformations, 330-334
- trees, 204
- types, 204
- validation, 81-82, 84-85
- viewing, 126
- well-formed, 81-82
- XML
 - creating*, 187
 - linking CSS*, 165-166
 - modeling*, 100-104
 - structure*, 63
- DoGet() method, 477-478**
- DOM (Document Object Model), 35, 198**
- browsers, 199, 225-229
- databases, 229
- documents
 - creating*, 279-281, 283, 292
 - creating without*, 283-288
 - types*, 204
- editors, 229
- IBM parsers, 224
- interface, 232-233
- Java, 221-223, 482
- JavaScript, 199-202
- levels, 198
- nodes, 202-203
- OMG IDL, 220-221
- SAX, 35, 231
- state, 207-210, 276
- Text object, 206-207
- trees, 236
- domain names, 115-116**
- DoPost() method, 477**
- DoPut() method, 477**
- double primitive type, 465**
- downloading**
 - GMD-IPSI engines, 155, 229
 - HTML tags, 7
 - images, 324
 - Java, 84, 459-461
 - JDK, 241
 - Jetty, 381
 - LotusXSL, 381
 - Notepad, 61
 - parsers, 192
 - standalone documents, 80
 - XCommerce, 349-351
 - XML, 84, 381
- DTDs (Document Type Definitions), 69**
- attributes, 75-76, 96-97
- automating, 96
- browsers, 296
- conditional sections, 91
- connectors, 73-74
- content model, 71, 74
- conversion, 322
- creating, 97
- default, 76, 304
- designing, 91-93
- documents, 79-81
- editors, 71-72, 74, 81-83, 97
- elements, 96-97
- entities, 80, 95
- events, 242
- existing, 92
- extensions, 97
- functions, 296
- inheritance, 105
- interface, 246
- keywords, 72
- limitations, 105
- managing, 90
- namespaces, 119-120
- nested, 93
- occurrence indicators, 73
- online help, 104
- owners, 94
- properties, 95
- public identifiers, 78-79
- repetition, 105
- root, 93-94
- schemas, 105
- SGML, 18-19, 105
- sharing, 321
- standards, 296
- structural information, 84, 98-99, 336-339
- subsets, 77-79
- support, 296
- syntax, 70-71, 105
- top-level, 93
- trees, 95
- UML, 92
- URIs, 96

WYSIWYG editors, 184
 XSLT, 128, 296, 303
Dump service, 293
duplication of namespaces, 112-113
dynamic invocation, 357
dynamic servlets, 472

E

EDI (Electronic Data Interchange), 64, 322-323

editors, 37, 60-61

CSS, 182-183
 documents, 276
 DOM, 229
 DTDs, 81, 97, 184
 elements, 341
 parsers, 219
 SGML, 37
 style sheets, 148-149
 text, 183
 tree-based, 183
 WYSIWYG, 184

Electronic Data Interchange (EDI), 64, 322-323

electronic markup, 14-15

elements, 96-97, 290
 attributes, 46-47, 75, 244-245
 browsers, 341
 common, 319-320
 content, 339-340
 declaration, 71-72
 defining, 80, 162
 documents, 11-14, 82-83, 203, 265-266
 editors, 47, 72, 341
 end tags, 44-45, 244
 HTML, 80
 indenting, 74
 links (XLink), 324
 names, 45-46, 60, 103-104

namespaces, 314-315
 nesting, 47-48, 93, 340-341
 nodes, 203
 objects, 206, 278
 owners, 94
 parsers, 244, 248
 properties, 342
 reducing, 335-336
 repeating, 341
 reusing, 342
 roots, 48-49, 93-94
 selecting, 166
 signatures, 328
 start tags, 44-45, 244
 state, 261, 264
 subelements, 319
 text, 72
 Text object, 206
 top-level, 77, 93, 138
 trees, 204-206
 XSLT, 128, 134, 140, 149, 154-155
email, 346-347
emailaholic directory, 318, 383
empty elements, 47, 72
encapsulation, 417-428
encoding
 declarations, 51
 end tags, 44-45, 59
 parameters, 51-52
 source codes, 56-57
 UTF, 51
eNotepad, 183
entities, 52
 accessing, 85
 characters, 53
 declaration, 80
 defining, 52
 documents, 85
 DTDs, 80, 90
 external, 86-89
 general, 86

HTML, 153
 internal, 86-89
 parameters, 86-87, 95
 parsed, 86-87
 references, 52
 resolution, 242, 246
 standalone documents, 79
 storing, 85
 unparsed, 86-90, 246
enumerated type attributes, 75
errors
 Document object, 204
 Java, 470-471
 messages, 132, 242, 246
 parsers, 218-220
 SAX, 246-247
escapeXML() function, 291
events
 generating, 234-236
 handlers, 233, 242
 interface, 233-236
 parsers, 196-197, 242-244, 248
see also SAX
exceptions, Java errors, 470-471
exchange formats, 356
exchange rate attribute, 211, 216-217
existing DTDs, 92
exportProduct() function, 303
extended links (XLink), 326-327
extends keyword (Java), 477
extensibility, 308
eXtensible Markup Language (XML), 6
extensions, 97, 320
external entities, 86-90
external subsets, 77-80

extracting data, 155-158
extranets, 31

F

fatalError() method, 246
fields in Java, 466
files, configuration, 382-383
filters
 middle tier, 369
 ratings, 111
fixed attributes
 namespaces, 119-120
 values, 84
Flash, 6
flexibility
 documents, 265-266
 XSLT, 296
float primitive type, 465
flow objects, 168
 boxes, 169-172
 CSS, 189
 Java, 464-465
 properties, 169
fonts, 176-178
forcing document structure, 266
foreground color, 178
formats
 common, 357-359
 conversion, 29
 DTDs, 303
 electronic markup languages, 15
 HTML, 24-25
 XSLFO, 189
 XSLT, 304
frames, 225
functions
 DTDs, 296
 XSLT, 136-137

G

general entities, 86
generating
 events, 234-236
 parsers, 269-270
 XCommerce, 351-353
 XSLT style sheets, 132-133
generic coding
 HTML, 24
 identifier (GI), 17
getAttribute() method, 278
getColumnNumber() method, 245
getLength() method, 245
getLineNumber() method, 245
getName(i) method, 245
getNamedItem() method, 217
getPublicId() method, 245
getSystemId() method, 245
getTopLevel() function, 282
getType(i)/getType(name) method, 245
getValue(i)/getValue(name) method, 245
global classpaths, 480
GMD-IPSI engines, 155, 229
Goldfarb, Charles (Dr.), 18
graphics, 349
grouping font properties, 178

H

HandlerBase class, 242
hasChildNodes() method, 278

help tools, 104
hiding syntax, 290-291
hierarchy, XSLT templates, 141
highlighting text, 183
Hotmail, 347
HTML (Hypertext Markup Language)
 attributes, 21-23
 client/server applications, 358
 conversions, 332
 data structure, 284
 documents, 152, 292-294
 elements, 80
 entities, 153
 formatting, 24-25
 generic coding, 24
 limitations, 7, 162
 popularity, 6
 procedural markup, 24
 structural information, 21
 style sheets, 132-133
 syntax, 333-334
 tags, 6-7, 20-21
 XSLT, 128, 297
HTTPPost class, 414-417
hyperlinks, 189

I

IBM, 224-225
identifiers, 78-79
IETF (Internet Engineering Task Force), 117
images
 background, 178-179
 downloading, 324
importing
 classes, 468
 style sheets, 182
InDelv browser, 187-189
InDelv XML Browser, 37

indenting

elements, 74
 source codes, 55-56
 text, 177

indexes, 185**indicators, occurrence, 73****inheritance**

DTDs, 105
 flow objects, 169
 Java, 476-477, 479-480

inline boxes, 171**InputSource interface, 243****insertBefore() function, 276-277****insertData() method, 278****instructions, SAX, processing, 244****int primitive type, 465****interface**

applications with parsers, 194, 196
 DTDHandler, 246
 EntityResolver, 246
 ErrorHandler, 246
 event-based, 196-197, 233-236
 IDL, 220
 InputSource, 243
 Java, 479-480
 object-based, 232-233, 236
 ParserFactory, 243
 SAX, 231, 246-247, 260, 479
 vendors, 197

internal entities, 86-89**internal subsets, 77-79****international standards**

SGML, 18
 Unicode, 51

Internet Engineering

Task Force (IETF), 117, 328

Internet Explorer

channels, 65-66
 DOM support, 276
 links (XLink), 325
 style sheets, 145-149, 302
 support, 36, 372
 XML, 129-130
 XSLT, 147-148, 296

InterNIC, 115**intranets, 308-309, 375****islands, 202, 276****J****Java, 6, 8, 48, 377-378**

access control, 468-469
 applications, 458
 archives, 481-482
 classes, 459, 466
 classpaths, 475, 480-481
 code, 462-463
 command-line version, 84
 comments, 469-470
 constructors, 467
 conversion utility, 221-223
 core APIs, 482
 Development Kit (JDK), 127, 241, 459-460
 errors, 470-471
 fields, 466
 files, 383-384
 flow, 464-465
 imports, 468
 inheritance, 476-480
 interfaces, 479-480
 Java Runtime Environment (JRE), 85, 127, 459
 Javadoc, 469-470
 JavaScript, 223-224
 keywords, 471, 477-478
 methods, 467, 477-478
 objects, 466
 packages, 468
 parameters, 467
 parsers, 221, 460
 price list, 249-260

servlets, 473-475

downloading, 460-461
properties, 475-476

tools, 459

variables, 465-467

Virtual Machine (JVM), 459

Web Server, 461

XCommerce, 349

XML, 237

see also SAX

JavaBean, 294**JavaScript, 6, 8, 48, 376-377**

clients, 373, 375
 code, 272-274
 conversion.js, 200-201
 documents, 280-281, 284-288, 295-296
 DOM application, 199-202
 Java, 223-224
 methods, 289
 object constructors, 289
 XSLT, 298-301

JDBC, 382**JDK (Java Development Kit), 127, 241, 459-460****Jetty, 381, 461****JRE (Java Runtime Environment), 127****JRun, 461****K-L****keywords**

DTDs, 72
 Java, 471, 477-478

languages, 53, 376**length property, 172-173****libraries, 381-382, 384****limitations, 105****line height, 177****linking, 10, 165**

documents, 165-166
 style sheets, 276

- templates, 313
- XLink
 - adding*, 326
 - attributes*, 324
 - browsers*, 327
 - elements*, 324
 - extended*, 326-327
 - Internet Explorer*, 325
 - simple*, 323-325
 - storing*, 326
- listings**
- address books, 42-43,
70-71, 98-99, 102-103
- articles, 129-130
- attribute conversion, 212,
214-216
- catalog files, 79
- channel definitions, 65-66
- conditional sections, 91
- configuration files,
382-383
- conversion utility, 212,
221-223, 225-227
- conversion.js, 200-201
- CORBA, 356-357
- country entities, 88
- CSS
 - boxes*, 169-170
 - linking XML documents*, 165-166
 - simple*, 164
 - style sheets*, 179-180,
227-228
 - tables of contents*,
185-186
- currency, 199-200, 211
- documents, 27-28, 100,
330-331
- DOM, 270-271, 279
- DTDs, 82, 94-96, 337
- element names, 46
- Emailaholic style sheet,
368-369
- end tags, 56
- entries, 77
- exchange rate attribute,
211
- external entities, 86-87,
89
- extracting data, 156-157
- frames, 225
- HTML
 - attributes*, 21-23
 - code*, 20-21, 24-25, 297
 - conversions*, 332-333
 - documents*, 132-133,
152, 284
 - results*, 333-334
 - sending document to
server*, 293
 - writing*, 363-364
- indexes, 157-158
- Internet Explorer,
145-148, 302
- invalid document, 83
- Java
 - application*, 462-463
 - Checkout class*, 407-414
 - Comlet class*, 390-393
 - Editor class*, 444-446
 - HTTPPost class*,
414-417
 - Merchant class*, 398-403
 - MerchantCollection
class*, 393-396
 - NotImplementedError
class*, 428
 - Product object*, 404-407
 - Shop class*, 386-390
 - Viewer class*, 451-454
 - XMLServer class*,
429-434
 - XMLServerConsole
class*, 435-444
 - XMLUtil class*, 417-427
- JavaScript, 280-281,
284-288, 373-374,
447-450
- links , 35
- memo, 10
- merging files, 156
- names, 58
- namespaces, 33, 121-122
 - attributes*, 118-120
 - different names*, 111-112
 - duplication*, 112-113
 - prefix declaration*, 114,
119
 - ratings*, 109-111
 - scoping*, 118
- newsletters, 62-63
- orders, 63-64
- phone lists, 308-310
- price lists, 195, 234,
248-260, 312-316
- product lists, 31, 359-360,
366-367, 370-371
- resources, 108-109
- RTF, 15-16
- SAX, 237-241, 479
- Server Configuration File,
475
- Service-Side JavaScript,
376
- servlets, 473-476
- SGML, 19
- state information, 208-209
- style sheets, 34-35,
149-151
 - alternate*, 310-311
 - combining*, 320
 - common elements*,
318-319
 - conversions*, 131-132
 - Editor class*, 450-451
 - Emailaholic*, 318
 - Merchants*, 397
 - Playfield*, 316-317
 - Viewer class*, 454-455
 - WriteIT*, 317
 - XMLi*, 317
- TeX, 17
- text style sheet, 142-143
- valid documents, 82-83
- valid invoices, 338

Web servers, 294-295
writing, 361-363
XCommerce, 359-361, 365
XSLFO, 187-189

loading

style sheets, 320
XSLT, 305

location

elements in documents,
82-83
state, 261
XSLT templates, 138-139

Locator object, 245**Locomotive, 461****long primitive type, 465****LotusXSL, 37, 132, 310,**

320, 381, 460

browsers, 296
processors, 127
XSLT style sheets, 132

M**macros, 17-18****maintaining**

DOM
state, 208-210
SAX, 247-260

makeXML() function, 289**managing DTDs with entities, 90****mapping, 220, 322****margins, boxes, 174-175****markup languages**

comments, 50
documents, 334-336
electronic, 14
formatting, 15
plain text, 43
procedural, 15
readability, 43
software, 44
source codes, 14

structure, 43
tags, 17

matching XSLT items, 136-139**Megginson, David, 237****Merchant class, 397-403****MerchantCollection class, 315-316, 393-397****merging data extracts, 156-157****messages, 219-220****methods**

ActiveX, 294
CharacterData, 278
Document, 277
Element, 278
Java, 467, 477-478
JavaScript, 289
NamedNodeMap object,
217
Node, 277-278
SAX, 243
Text, 279

Microsoft, 224, 460**middle tier application, 347**

clients, 372, 374-375
cost, 366
customizing, 367
filters, 369
style sheets, 369-371
tools, 366
XCommerce, 349, 386-393

middleware

applications, 356
CORBA, 356-357
dynamic invocation, 357
protocols, 356
tools, 356

modeling XML documents, 100-104**Mozilla, 36, 145****MP3, 6****multiple inheritance, Java, 479-480****multiple items, selecting, 154-155, 166****N****n-tiered applications, 345**

adding, 348
client/server, 346-347
three-tiered, 347-348

NameNodeMap object, 217**names**

attributes, 217
elements, 45-46, 103-104
fonts, 176
nodes, 203, 217
parent, 203
spaces, 60
syntax, 58

namespaces

conflicts, 108, 112
declaration, 114, 120
domain parking, 116
DTDs, 119-120
duplication, 112-113
elements, 120, 314-315
fixed attributes, 119-120
names, 114
prefixes, 113-114
PURLs, 118
ratings, 108-111
scoping, 118
style sheets, 121-122
TLDs, 115
URLs, 114-115, 117
URNs, 117-118
Xlink, 122
XML resources, 108-109

Near & Far, 104**nesting elements, 47-48, 93, 340-341**

Netscape**Communicator, 36****newsletters, 62-63****nodes**

adding, 274-276

attributes, 202

child, 202, 275

codes, 203

creating, 281-282

DOM, 202-203

elements, 203

names, 217

objects, 206-207, 217

parent, 202

preceding, 202

properties, 277-278

trees, 205

types, 202

values, 203

XSLT functions, 136-137

nonvalidation parsers, 193, 225**non-XML data structures**

adding, 53-54

documents, 291-292

DOM, 288-289

nondestructive transformations, 332

notations

declaring, 246

unparsed entities, 89-90

Notepad, 60-61, 183**NotImplementedError class, 428****O****Object Management Group (OMG IDL), 220-221, 377****object models, 232-233**

adding, 275

attributes, 96-97, 218

automating, 96

constructors, 289

designing, 92-93

elements, 96-97

Java, 466

limitations, 236

nested, 93

owners, 94

parameter entities, 95

parsers, 194, 196, 243

properties, 95, 218

removing, 275

root, 93-94

top-level, 93

trees, 95

UML, 92

URIs, 96

XSLT, 140

occurrence indicators, 73**offloading databases, 7****OMG IDL (Object Management Group Interface Definition Language), 220-221, 377****open() method, 294****open-source software, 461****or connector, 338****Oracle, 37, 224****orders (XCommerce), 353-355**

forms, 63-64

generating, 351-353

processing, 365-366

output

DTDs, 303

XSLT style sheets, 134

owner elements, 94**P****package classes (Java), 468-469****padding boxes, 175****pages, printing, 168****parameters**

encoding, 51-52

entities, 86-87, 95

Java, 467

parents

attributes, 340

elements, 48

nodes, 202

parsers, 191, 231

attributes, 216-217

backend, 193

code, 224

compilers, 193

CORBA objects, 221, 296

documents, 204, 295

downloading, 192

elements, 244, 248

entities, 86-87

errors, 218-220

events, 196-197, 234-236

files, 243

generators, 269-270

IBM, 224-225

Java, 221, 460

Microsoft, 224

nonvalidation, 193, 225

objects, 194, 196, 204,

243-244

Oracle, 224

parse() method, 243

ParserFactory interface, 243

SAX, 242, 246-247

standards, 197-198

Sun, 224

switching, 243

syntax, 37, 192-193

- validation, 193, 225
- XML architecture, 193-194
- paths**
 - length, 153
 - XSLT syntax, 135
- payments (XCommerce), 349**
- #PCDATA keyword, 72**
- PDAs, 6-7, 144**
- PDOM (Persistent DOM), 229**
- percentage property, 173**
- Perl, 237, 376**
- Permanent URLs (PURLs), 118**
- Persistent DOM (PDOM), 229**
- personal digital assistants (PDAs), 6**
- phone list style sheet, 309-310**
- platforms, 144, 219**
- Playfield style sheet, 316-317**
- plug-ins, grid posting to Web servers, 294**
- plus (+), 73**
- posting documents to Web servers, 293-294**
- preceding nodes, 202**
- predefined tags, 9**
- prefixes, namespaces, 113-114**
- presentation tier (three-tiered application), 347**
- price lists**
 - attributes, 216-217
 - comparison, 312-313
 - merchants, 315-316
- print servers, 346**
- printing pages, 168**
- priorities, CSS, 167-168**
- private classes, 469**

- procedural markup**
 - HTML, 24
 - RTF, 15-16
- processing**
 - instructions (PI), 53-54, 244
 - XCommerce orders, 349, 365-366
- products (XCommerce)**
 - databases, 361-364
 - downloading, 349-351
 - listing, 359-361
 - objects, 404-407
 - uploading, 364
- programming languages, 376**
 - Java, 377-378
 - JavaScript, 376-377
 - Omnimark, 377
 - Perl, 376
 - Python, 377
- ProjectX, 237, 460**
- properties**
 - adding, 95
 - attributes, 342
 - CharacterData, 278
 - CSS, 168, 172-173
 - directories, 383
 - documents, 277
 - elements, 278, 342
 - flow objects, 169
 - Java, 475-476
 - nodes, 277-278
 - protected classes, 469
 - text, 279
- protocols, middleware, 356**
- public classes, 469**
- public identifiers, 78-79**
- publishing application, 62-63**
- PURLs (Permanent URLs), 118**
- Python, 237, 377**

Q-R

- question mark (?), 73**
- quotes, attribute values, 60**
- ratings**
 - different names, 111-112
 - namespaces, 109-111
- RDF (Resource Definition Framework), 58**
- reading parser files, 243**
- recommendations**
 - namespaces, 108
 - XSL, 127
- recording state elements, 264**
- reducing elements, 335-336**
- references, 52-53**
- registration**
 - domain names, 115-116
 - event handlers, 242
 - TLDs, 116
- relational databases, 30**
- reloading databases, 7**
- Remote Procedure Call (RPC), see middleware**
- removing**
 - attributes, 278*
 - child from relationship, 277*
 - named items, 217*
 - objects, 275*
- rendering screens, 168**
- repeating elements, 341**
- repetition, DTD, 105**
- replaceChild() method, 277**
- replaceData() method, 278**
- Resource Definition Framework (RDF), 58**
- resources, namespaces, 108-109**

retrieving documents from Web servers, 293-295

reusing elements, 120, 342

root elements, 48-49, 93-94

RPC (Remote Procedure Call), *see* middleware

RTF (Rich Text Format), 15-16

S

saving documents, 276, 295-296

SAX (Simple API for XML), 35, 198

architecture, 260-261

AttributeList, 244-245

classes, 242, 260

code, 237-241

compiling, 241-242

documents ending, 244

elements event handlers, 242-244

IBM parsers, 224

instructions, 244

interface, 246, 260, 479

Java, 237

Locator object, 244-245

methods, 243

notations, 246

parsers, 243, 246-247

Perl, 237

ProjectX, 237

Python, 237

state

efficiency, 265

elements, 261

location, 261

maintaining, 247-260

recording elements, 264

transition values,

262-265

updating, 264

trees, 263

scalability, 358

schemas, 105

scientific applications, 7

scoping namespaces, 118

screens, 168, 376

scripting languages, 458

search engines, 6

security, 6, 349

selectors, 166-167

semantics, 58, 64-65

send() method, 294

sending documents to the server, 292-294

server-side applications

CGI, 473

components, 458-459

documents, 292-295

Java, 377-378, 458

JavaScript (SSJS),

376-377

Omnimark, 377

Perl, 376

Python, 377

see also client/server applications

servlets

classpaths, 475

downloading, 460-461

dynamic, 472

Java, 473-475

properties, 475-476

writing, 293

settings, *see* configuration

SGML (Standard Generalized Markup Language)

CALS, 20

DocBook, 20

documents, 18

DTDs, 18-19, 105

editors, 37

international standards, 18

models, 18

syntax, 19

sharing documents, 321

shell scripts, 85

shopping carts (XCommerce), 349

short primitive type, 465

sibling selectors (CSS2), 181

signatures

attributes, 328

digital, 328-329

elements, 328

standards, 327

Simple API, *see* SAX

simple CSS, 164-165

simple links (XLink), 323-325

sizing fonts, 176

smart phones, 6-7

software

open-source, 461

speed, 44

source code

Document object, 204

HTML, 297

indenting, 55-56

Java, 462-463

JavaScript, 272-274

mark-up, 14

nodes, 203

parsers, 224

SAX, 237-241

trees, 153

SOX (Schema for Object-oriented XML), 105

spaces

attributes, 53

element names, 60

speed, 44

splitText() method, 279**SQL language, 44****standalone documents**

downloading, 80

DTD, 79-80

entities, 79

external subsets, 80

**Standard Generalized
Markup Language, see
SGML****standards**

applications, 197-198

CALs, 20

companions, 32

CSS, 33-35

databases, 197

DTDs, 296

EDI, 322

international, 18

namespaces, 33

parsers, 197-198

schemas, 105

signatures, 327

syntax, 10

Unicode, 50-52

vCard, 101-102

W3C, 32

XLink, 35-36, 323

XPointer, 35-36

XQL, 155

XSL, 33-35

start tags, 44-45**start() event, 244****Startmail, 347****state**

DOM, 207-210

elements, 261, 264

location, 261

SAX

*efficiency, 265**maintaining, 247-260**transition values,**262-265*

updating, 264

statements, 53-54**static modifier, 467****storage**

attributes, 339-340

documents, 229

elements, 339-340

entities, 85

links (XLink), 326

XCommerce, 353-355

streaming tags, 6**string attributes, 75****structural information**

attributes, 218

documents, 10-14, 18, 266

DTDs, 84, 98-99, 303,
336-339

HTML, 21

markup, 43

non-XML, 291-292

tree-like, 42

XSLT source trees, 153

style sheets, 10, 318

advanced, 179-180

alternate, 310-311

applying, 145-148

attaching, 276

combining, 319

common elements, 319,
367

data extracts, 157-158

documents, 331

editing, 148-149

Emailaholic, 318

fonts, 177

HTML, 153

importing, 182

Internet Explorer, 302

loading, 320

middle tier, 369-371

namespaces, 121-122

phone lists, 309-310

Playfield, 316-317

text, 141-144

WriteIT, 317

XMLi, 317

XSLT, 304-305

*automating, 305**conversion, 131-132**creating, 149-151**elements, 134**generating, 132-133**LotusXSL, 132**output, 134**see also CSS***subelements, 319****subscriptions to Web
sites, 65-66****subsets, 77-79, 263****substringData() method,
278****Sun**

parsers, 224

ProjectX, 237

**super keyword (Java),
478****support**

DOM, 276

DTDs, 296

Internet Explorer, 372

switching parsers, 243**syntax**

attributes, 46-47, 60

case-sensitive, 60

CDATA sections, 54-55

character data, 43

DTD, 70-71, 105

elements, 45-46

end tags, 44-45, 56-57, 59

entities, 52-53

hiding, 290-291

HTML, 333-334

markup, 43-44

parsers, 37, 192-193

rules, 59

SGML, 19

source codes, 55-56

spaces, 60

standards, 10

start tags, 44-45

Unicode, 50-52
 XSLT paths, 135
system identifiers, 78

T

tables of contents, 185-186

tags

development, 8
 HTML, 6-7, 20-21
 macros, 17-18
 markup languages, 17
 names, 206
 predefined, 9
 search engines, 6
 security, 6
 specialized, 8
 streaming, 6
 XML, 27-28

targets, processing instructions (PI), 54

templates

calling, 154
 links, 313
 XSLT
built-in, 138-139
elements, 134
hierarchy, 141
locating, 138-139
matching, 137, 139-140

text

alignment, 177
 conversion, 141-144
 editors, 183
 elements, 72
 indent, 177
 line height, 177
 objects, 206-207
 plain, 43
 properties, 279

this keyword (Java), 478

three-tiered applications, 347-348

throw keyword (Java), 471

TLD (Top Level Domain), 93, 115-116, 138, 282-283

tokenized attributes, 75

tools

Java, 459
 middle tier, 366
 middleware, 356
 XCommerce, 417-428

Top Level Domain (TLD), 93, 115-116, 138, 282-283

toXML() function, 289

transformations in documents, 330-334

transformNode() method, 304

transitions, SAX state, 262-265

trees

Document object, 204
 DOM, 236
 editors, 183
 elements, 204-206
 objects, 275
 structures, 42
 subsets, 263

types, documents, 202, 204

U

UML (Unified Modeling Language), 92

underscore (), 45

Unicode, 50-52

unparsed entities, 86-90

updating

address books, 308-309, 312
 databases, 32

phone lists, 312
 state, 264

uploading XCommerce products, 364

URIs (Universal Resource Identifiers), 78, 96

URLs (Uniform Resource Locators), 384

creating, 114, 117
 domain names, 115
 flexibility, 115
 namespaces, 114-115
 properties, 173

URNs (Uniform Resource Names), 117-118

UTF encodings, 51, 81-82

V

validation

batch files, 85
 documents, 84-85
 parsers, 193, 225
 shell scripts, 85

values

attributes, 60, 75-76, 84, 217, 341
 default, 80
 nodes, 203
 SAX, 262-265

variables, 465-467

vCard, 43, 101-102

vendor interface, 197

vertical bar (|), 73-74

viewing

documents, 126
 extensions, 320
 Web sites, 144
 XML in browsers, 129-130

W

W3C (World Wide Web Consortium), 6, 32, 105
walking
 nodes, 210
 products, 210, 246
 tree elements, 204-206

Web sites
 access, 6
 designing, 26-27
 development, 6
 EDI, 323
 Hotmail, 347
 InterNIC, 115
 Microsoft, 224
 Near & Far, 104
 Omnimark, 377
 Oracle, 224
 platforms, 144
 servers, 293-295
 Startmail, 347
 subscriptions, 65-66
 Sun, 224
 Unicode, 50
 viewing, 144
 W3C, 6
 XML links, 92
 XTransGen (IBM), 305

Webmail, 347
WebSTAR, 461
weight, fonts, 177
well-formed documents, 81-82
whitespace, parsers, 244
World Wide Web Consortium (W3C), 6, 32, 105
WriteIT style sheet, 317
writing
 applications, 356
 servlets, 293
 XML, 289-290

WYSIWYG editors, 184

X-Z

XCommerce, 348
 checkout, 351, 407-417
 classpaths, 381-382, 384
 common formats, 358-359
 configuration files, 382-383
 data tier, 353-355
 databases, 361-364, 384-385
 directories, 383
 Editor class, 444-446
 graphics, 349
 HTTPPost class, 414-417
 Java files
 classes, 349
 compiling, 383-384
 JavaScript, 447-450
 Merchant class, 397-403
 MerchantCollection class, 393-397
 middle tier, 349
 clients, 372-375
 Comlet class, 390-393
 cost, 366
 customizing, 367
 filters, 369
 Shop class, 386-390
 tools, 366
 NotImplementedError class, 428
 orders, 351-353, 365-366
 payments, 349
 products
 downloading, 349-351
 listing, 359-361
 objects, 404-407
 uploading, 364
 security, 349
 shopping carts, 349
 style sheets, 369-371, 450-451
 tools, encapsulation, 417-428

Viewer class, 451-455
 XMLServer class data tier, 429-434
 XMLServerConsole class data tier, 435-444
 XMLUtil class, 417-427

XHTML, 144
XJParse, 237, 460
XLink, 35-36, 323
 adding links, 326
 attributes, 324
 browsers, 327
 elements, 324
 extended, 326-327
 Internet Explorer, 325
 namespaces, 122
 simple, 323-325
 standards, 323
 storing, 326

XMetaL, 37, 54, 184
XML (eXtensible Markup Language), 6
XML for Java, 237, 381, 460
XML-Data, 105
XMLi
 Editor class, 444-446
 JavaScript, 447-450
 style sheets, 317, 450-451
 Viewer class, 451-455
 XCommerce
 directories, 383
XMLUtil class, 417-427
XP (James Clark), 237
XPointer, 35-36
XQL (XML Query Language), 155
XSL (XML Stylesheet Language), 33-35
 directories, 383
 documents, 126
 processors, 37, 127
 recommendations, 127
 style sheets, 148-149

XSLFO (XSL Formatting Objects), 127, 161, 187-189

XSLT (XSL

Transformation), 127

advantages, 304

attributes, 136

calling, 303-304

CSS, 162-163, 185-187

data extracts, 155-158

DTDs, 128, 296, 303

elements, 128, 140, 149, 154-155

errors, 220

flexibility, 296

formats, 304

functions, 136-137

HTML, 128, 152-153, 297

Internet Explorer, 296

Internet Explorer 5.0, 147-148

JavaScript, 298-301

loading, 305

objects, 140

paths, 135, 153

source trees, 153

style sheets, 304

automating, 305

conversion, 131-132

creating, 149-151

elements, 134

generating documents, 132-133

LotusXSL, 132

output, 134

templates

built-in, 138-139

calling, 154

elements, 134

hierarchy, 141

locating, 138-139

matching, 137, 139-140

XTransGen, 305